



Projeto Brasil 6G Fase III

Atividade 4.5 – Implantação e disponibilização de infraestrutura de HPC - Parte 2



Histórico de Atualizações:

Versão	Data	Autor(es)	Notas
1	03/07/2026	Bruno Ciro do Nascimento (RNP) Glauber Soares dos Santos (RNP)	Elaboração de conteúdo
2	13/07/2026	Diego Gabriel Soares Pivoto Giovana Pereira Palma Juliano Silveira Ferreira	Revisão inicial
3	28/07/2026	Bruno Ciro do Nascimento (RNP) Glauber Soares dos Santos (RNP)	Complemento de conteúdo
4	30/07/2026	Luciano Leonel Mendes Vítor Rodrigues Di Toro	Revisão geral

Lista de Figuras

1	Pilha de <i>softwares</i> e serviços de HPC e IA (Gerado por IA).	2
2	Ilustração dos componentes do <i>Slurm</i> (Gerado por IA).	5
3	Ilustração dos componentes do K8S (Gerado por IA).	7
4	Ilustração dos componentes do Run:ai (Gerado por IA).	9
5	Ilustração da solução NVAIE.	12
6	Interface <i>web</i> do gestor BCM.	13
7	Estrutura geograficamente distribuída.	14
8	Arquitetura integrada do Run:ai.	16
9	Fluxo de treinamento.	17
10	Fluxo de inferência.	17
11	Ilustração das camadas de virtualização e containerização.	27
12	Listando os componentes do <i>cluster</i> K8S criado.	30
13	<i>Deploy</i> do NVIDIA <i>device plugin DaemonSet</i> .	32
14	Recursos de GPU disponibilizados de forma granular.	34
15	Múltiplos <i>pods</i> em execução em uma mesma GPU.	35
16	<i>Pods</i> em execução com fatias de tamanhos diferentes.	36
17	Componentes do JupyterHub no <i>Kubernetes</i> (K8S).	37
18	Interface <i>web</i> do JupyterHub.	38
19	Arquitetura do Slinky (Gerado por IA).	42
20	Interface do <i>Base View</i> .	44
21	Situação dos dispositivos incluídos.	45
22	Informações sobre a licença.	46
23	Exemplo de instalação do <i>Edge Director</i> .	47
24	Visão do dispositivo de borda ativo.	47
25	Perfis MIG criados em GPU DGX H200.	48
26	Interface do K8S <i>deploy Wizard</i> .	49
27	GPU Operator e componentes em execução no <i>cluster</i> .	50
28	Interface gráfica para <i>deploy</i> do JupyterHub.	51
29	NVIDIA BasePOD: Arquitetura de referência.	53
30	Arquitetura da solução computacional.	55
31	Rede externa conectada ao <i>Head Node</i> a 100 Gbps.	57
32	Rede interna interconectando todos os nós a 200 Gbps.	57
33	Visão frontal dos racks em processo de montagem - 1 à esquerda e 2 à direita.	58
34	Visão traseira dos racks em processo de montagem - 1 à esquerda e 2 à direita.	59
36	Ferramentas que serão disponibilizadas para os pesquisadores.	59
35	Portal de acesso ao BCM, instalado no <i>Head Node</i> .	60
37	Portal de acesso ao Run:ai, com 1 GPU <i>node</i> disponível.	61

Lista de Tabelas

1	Especificações das VMs do <i>cluster</i> Kubernetes.	26
---	--	----

Acrônimos

6G Sexta Geração de Rede Móvel Celular

AI *Artificial Intelligence*

API *Application Programming Interface*

AMD-Vi *AMD Virtualization for Directed I/O*

BCM *Base Command Manager*

BTU *British Thermal Unit*

BIOS *Basic Input/Output System*

CI/CD *Continuous Integration/Continuous Deployment*

CIDR *Classless Inter-Domain Routing*

CLI *Command-Line Interface*

CND Centro Nacional de Dados

CNCF *Cloud Native Computing Foundation*

CNI *Container Network Interface*

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico

CPU *Central Processing Unit*

CRD *Custom Resource Definition*

CUDA *Compute Unified Device Architecture*

DCGM *Data Center GPU Manager*

DNS *Domain Name System*

DHCP *Dynamic Host Configuration Protocol*

TFTP *Trivial File Transfer Protocol*

FLARE *Federated Learning Application Runtime Environment*

FNDCT Fundo Nacional de Desenvolvimento Científico e Tecnológico

Gbps *Gigabits Per Second*

GID *Group Identifier*

GPU *Graphics Processing Unit*

GRES *Generic Resources*

GRUB *GRand Unified Bootloader*

HPC *High-Performance Computing*

IA Inteligência Artificial

ICMP *Internet Control Message Protocol*

IDE *Integrated Development Environment*

IOMMU *Input/Output Memory Management Unit*

Inatel Instituto Nacional de Telecomunicações

IoT *Internet of Things*
IP *Internet Protocol*
K8S *Kubernetes*
KVM *Kernel-based Virtual Machine*
LVM *Logical Volume Manager*
LLM *Large Language Model*
MAC *Media Access Control*
ML *Machine Learning*
MLOps *Machine Learning Operations*
MPI *Message Passing Interface*
MIG *Multi-Instance GPU*
MUNGE *MUNGE Uid 'N' Gid Emporium*
NFD *Node Feature Discovery*
NVAIE *NVIDIA Artificial Intelligence (AI) Enterprise*
NGC *NVIDIA GPU Cloud*
OIDC *OpenID Connect*
PCIe *Peripheral Component Interconnect Express*
PXE *Preboot Execution Environment*
POC *Proof of Concept*
PBS *Portable Batch System*
PDS *Policy Driven Scheduling*
PDU *Power Distribution Unit*
PVE *Proxmox Virtual Environment*
PVC *Persistent Volume Claim*
QoS *Quality of Service*
RAM *Random Access Memory*
RBAC *Role-Based Access Control*
RDMA *Remote Direct Memory Access*
REST *Representational State Transfer*
RNP *Rede Nacional de Ensino e Pesquisa*
RoCE *RDMA over Converged Ethernet*
SAML *Security Assertion Markup Language*
SHA256 *Secure Hash Algorithm 256-bit*
SSH *Secure Shell*
TR *Termo de Referência*
TRES *Trackable Resources*

UID *User Identifier*

VRAM *Video Random Access Memory*

VT-d *Virtualization Technology for Directed I/O*

VFIO *Virtual Function I/O*

WAN *Wide Area Network*

WLM *Workload Manager*

YAML *Yet Another Markup Language*

Sumário

1	Introdução	1
2	Panorama Tecnológico	4
2.1	Gerenciadores de Cargas de Trabalho (HPC Tradicional)	4
2.1.1	Slurm	4
2.1.2	PBS Pro	5
2.2	Containerização em HPC	6
2.2.1	Apptainer (Singularity)	6
2.3	Orquestradores de Contêineres (<i>Cloud-Native</i>)	6
2.3.1	Kubernetes	6
2.4	Escalonadores Avançados e <i>Middleware</i> para Kubernetes	7
2.4.1	Volcano	7
2.4.2	Slinky	8
2.4.3	Run:ai	8
2.5	Plataformas de MLOps e Interfaces de Usuário	8
2.5.1	JupyterHub	9
2.5.2	Kubeflow	9
2.5.3	Alnoda	10
2.6	Orquestração de Aprendizado Federado	10
2.6.1	Flower	10
2.6.2	NVIDIA FLARE	11
3	Arquitetura Proposta: <i>Suíte NVIDIA AI Enterprise</i>	12
3.1	Provisionamento e Gerenciamento com o BCM	13
3.2	Orquestração e Extensão para a Borda	13
3.3	Infraestrutura CNCF: NVIDIA GPU <i>Operator</i>	15
3.4	NVAIE e Bibliotecas Otimizadas	15
3.5	Integração da Camada de MLOps no Ecossistema	15
3.5.1	Fluxo de Trabalho Interativo	16
4	POC e Análise de Alternativas	19
4.1	Arquitetura e Escopo	19
4.2	Cenário A: HPC Tradicional	19
4.2.1	Fundação de <i>Softwares</i> e Aceleração por GPU	20
4.2.2	Malha de Confiança e Autenticação	20
4.2.2.1	Sincronização de Usuários e Instalação	21
4.2.2.2	Geração da Chave Criptográfica em Nó mestre	21
4.2.2.3	Distribuição e Aplicação de Permissões	21
4.2.2.4	Inicialização e Validação do Serviço	21
4.2.3	Gerenciamento de Filas e Escalonamento	22
4.2.3.1	Instalação dos Pacotes do Controlador	22
4.2.3.2	Configuração Global (slurm.conf)	22
4.2.3.3	Inicialização do Controlador	23
4.2.4	Execução de Tarefas	23
4.2.4.1	Instalação e Sincronização de Configurações	23
4.2.4.2	Mapeamento Físico de GPU (gres.conf)	23

4.2.4.3	Inicialização e Validação do <i>Cluster</i>	24
4.2.5	Isolamento de Ambiente e <i>Runtimes</i> (Módulos e <i>Apptainer</i>)	24
4.2.5.1	Instalação do Lmod	24
4.2.5.2	Instalação do Apptainer (Singularity)	24
4.2.5.3	Uso Combinado na Submissão de Tarefas	25
4.3	Cenário B: Ecossistema <i>Cloud-Native</i> e Fatiamento Virtual	25
4.3.1	Virtualização da Infraestrutura (Proxmox)	26
4.3.1.1	Imagem base, CPU, memória e GPU	26
4.3.2	Mapeamento de GPU (<i>Passthrough</i>)	27
4.3.3	Orquestração de contêineres (Kubernetes)	28
4.3.3.1	Preparação do Sistema Operacional	28
4.3.3.2	Instalação do Contêiner <i>Runtime</i> e Componentes	29
4.3.3.3	Inicialização do <i>Control Plane</i> (Nó Mestre apenas)	29
4.3.3.4	Implantação do Container <i>Network Interface</i>	30
4.3.3.5	Ingresso dos Nós trabalhadores	30
4.3.4	Descoberta e Integração de GPU (NVIDIA <i>Device Plugin</i>)	31
4.3.4.1	Preparação do <i>Host: Driver</i> e <i>Container Toolkit</i>	31
4.3.4.2	Implantação do NVIDIA <i>Device Plugin</i> via <i>DaemonSet</i>	31
4.3.5	Gerenciamento de Filas e Fatiamento Virtual (Volcano)	32
4.3.5.1	Confirmação de Requisitos e Preparação	33
4.3.5.2	Instalação do Escalonador Central Volcano	33
4.3.5.3	Implantação do Volcano <i>Device Plugin</i>	33
4.3.6	Ambiente de Desenvolvimento Interativo (JupyterHub)	35
4.3.6.1	Preparação do Repositório e Configuração do <i>KubeSpawner</i>	36
4.3.6.2	Implantação e Exposição do Serviço	37
4.3.7	Orquestração de <i>Pipelines</i> de ML (<i>Kubeflow</i>)	39
4.3.7.1	Implantação via Kustomize	39
4.3.7.2	Sinergia com o Escalonador Volcano e GPU	39
4.3.8	Slinky	40
4.4	Cenário C: Arquitetura de Referência	41
4.4.1	Sistema Central de Gerenciamento (BCM)	42
4.4.2	Instalação e Provisionamento do Nó Controlador (<i>Head Node</i>)	42
4.4.2.1	Obtenção da Mídia e Requisitos Iniciais	43
4.4.2.2	Instalação e Configuração via <i>cm-installer</i>	43
4.4.2.3	Acesso Inicial e Administração via <i>Base View</i>	43
4.4.3	Descoberta e Adição Automatizada de Nós de Processamento	43
4.4.3.1	Descoberta de nós (<i>Node Discovery</i> via PXE)	44
4.4.3.2	Categorização e Atribuição de Imagem de <i>software</i>	44
4.4.3.3	Provisionamento Automático e Validação de Estado	44
4.4.4	Extensão da Infraestrutura para a Borda (<i>Edge Computing</i>)	45
4.4.4.1	Ativação e Requisitos de Licenciamento	46
4.4.4.2	Implantação do <i>Edge Director</i> via <i>Script</i>	46
4.4.4.3	Inclusão e Provisionamento do <i>Edge Node</i>	47
4.4.5	Particionamento Físico de Aceleradores (MIG)	47
4.4.5.1	Configuração Heterogênea do Nó (Exemplo Prático)	48
4.4.6	Implantação Automatizada do Kubernetes via BCM	49
4.4.6.1	Inicialização do Assistente	49

4.4.6.2	Definição de Topologia e Papéis	49
4.4.6.3	Configuração de Rede e Armazenamento	50
4.4.6.4	Integração Nativa com o Ecossistema NVIDIA (GPU <i>Operator</i>)	50
4.4.7	Portal Interativo de Desenvolvimento (JupyterHub)	50
4.4.7.1	Execução do Assistente de Instalação	51
4.5	Resultados Consolidados	52
4.5.1	Síntese dos Cenários Avaliados	52
5	<i>Status</i> Atual e Próximos Passos	53
5.1	Visão Geral do <i>Status</i> prévio	53
5.1.1	Equipamentos Adquiridos	54
5.1.2	Cotas de Importação e os Entraves da Ciência Brasileira	55
5.1.3	Local de Instalação e Entrega das Máquinas	56
5.2	Visão do <i>Status</i> Atual de Implantação do HPC	56
5.2.1	Instalação da Aplicação	57
5.3	Próximos Passos	60
5.3.1	Lições Aprendidas	61
5.3.2	Próximas POCs: Monitoramento, Bilhetagem e Autenticação Federada	62
6	Conclusão	63

1 Introdução

O projeto Brasil 6G tem como objetivo fomentar a pesquisa, o desenvolvimento e a validação de tecnologias voltadas às futuras redes de comunicação. Entre suas iniciativas estratégicas, destacam-se a especificação, a implantação e a disponibilização de uma infraestrutura de *High-Performance Computing* (HPC), objeto das atividades da atividade 4.5 e foco deste relatório.

A infraestrutura de HPC destina-se a apoiar a concepção, o desenvolvimento e a avaliação de desempenho de tecnologias emergentes para a Sexta Geração de Rede Móvel Celular (6G), oferecendo capacidade computacional para a execução de simulações em larga escala, o treinamento de modelos de Inteligência Artificial (IA) e o processamento intensivo de dados. Além de prover elevado poder computacional, sua concepção foi orientada para atender aos requisitos de uma infraestrutura compartilhada de pesquisa, permitindo o acesso remoto e o uso simultâneo de seus recursos por pesquisadores de diferentes instituições participantes do projeto. Dessa forma, busca-se promover a colaboração interinstitucional, ampliar o aproveitamento da infraestrutura computacional e potencializar os resultados científicos e tecnológicos produzidos no âmbito do Brasil 6G.

O presente relatório complementa a atividade executada na etapa anterior na qual foram definidos os componentes de *hardware* e a arquitetura física da infraestrutura de HPC, conforme detalhado em [1]. Na etapa atual de execução do projeto, as atividades relativas ao HPC tiveram como alvo principal a camada lógica da solução: a pilha de *softwares*, *frameworks* e bibliotecas que gerencia os recursos, orquestra as cargas de trabalho e viabiliza a oferta de serviços de IA e HPC aos pesquisadores, como ilustrado na Figura 1. Essa camada traduz a capacidade física instalada em serviços efetivamente utilizáveis pela comunidade científica e é, portanto, tão determinante para o sucesso do projeto quanto o próprio dimensionamento do *hardware*.

A conversão de poder de processamento bruto em capacidade prática de pesquisa depende da camada de orquestração escolhida. Esse domínio atravessa uma convergência entre os paradigmas tradicionais de HPC e o ecossistema *cloud-native*, o que abre múltiplos caminhos tecnológicos — desde suítes proprietárias integradas até arranjos construídos sobre ferramentas de código aberto. Levar qualquer um desses caminhos à operação sem validação prática prévia implica risco de subutilização do *hardware* adquirido, de aumento do esforço de sustentação e de descompasso com as práticas correntes de *Machine Learning Operations* (MLOps). O risco é agravado pela inércia da decisão: uma vez em produção, migrar de paradigma tem complexidade e custo operacional elevados. Soma-se a isso a maturidade técnica desigual das equipes de pesquisa, que torna a adequação de cada ferramenta sensível ao contexto organizacional, e não apenas às suas características técnicas.

Cabe delimitar, então, o que este trabalho decide e o que ele documenta. A arquitetura de referência da infraestrutura foi definida na etapa anterior do projeto: o levantamento de requisitos junto aos pesquisadores conduziu à adoção do NVIDIA BasePOD, e a formalização da compra dos equipamentos ocorreu em junho de 2025, conforme registrado na Seção 5. As *Proof of Concept* (POC) aqui relatadas foram conduzidas nesse intervalo, entre a definição da arquitetura e a instalação definitiva dos equipamentos. A pilha de *software* da arquitetura de referência, detalhada na Seção 3, decorre daquela definição e não é objeto de escolha neste relatório; os demais arranjos examinados entram como termos de comparação, e não como candidatos a substituí-la. O que permanecia em aberto era de outra ordem: verificar se a pilha adotada se comporta na prática como sua documentação descreve, mapear as competências necessárias para operá-la e dimensionar o que uma construção equivalente sobre ferramentas de código aberto exigiria da equipe.

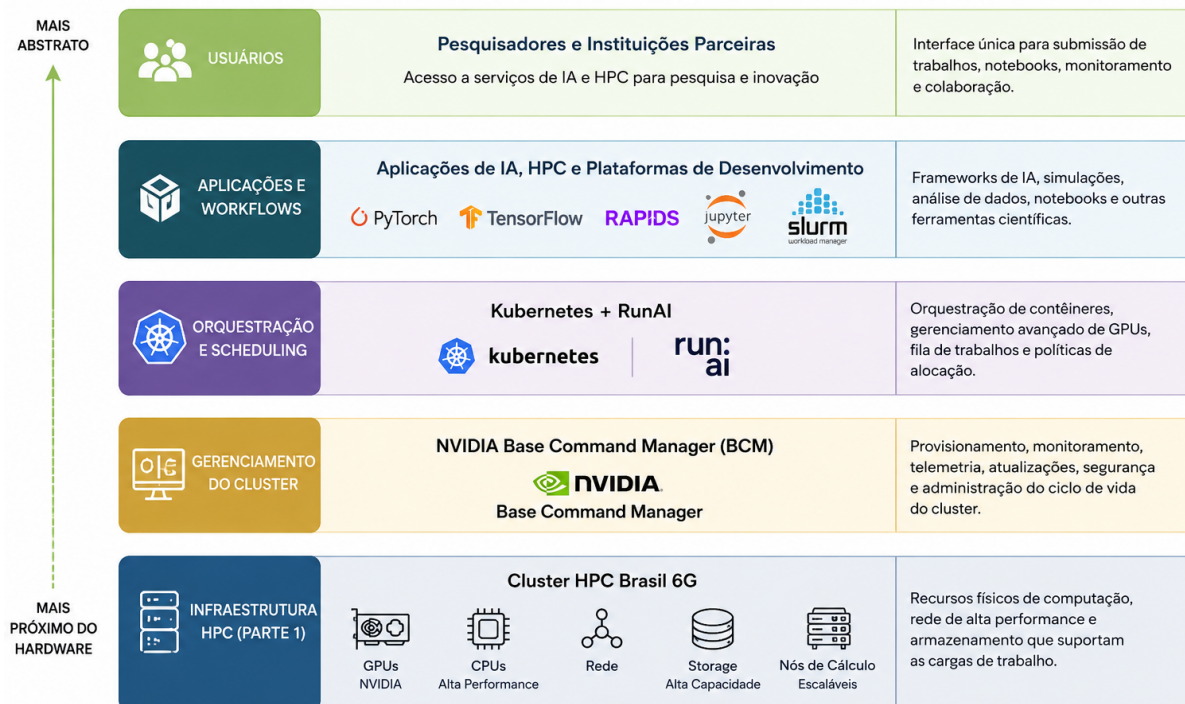


Figura 1: Pilha de *softwares* e serviços de HPC e IA (Gerado por IA).

Diante desse contexto, o objetivo geral deste trabalho é homologar, em ambiente virtualizado de prototipação, a pilha de *software* da arquitetura de referência adotada e situá-la frente às alternativas de código aberto equivalentes. Documenta-se, para cada caminho, o esforço operacional envolvido e as competências técnicas que ele exige da equipe. Como objetivos específicos, o relatório propõe-se a:

- Levantar o panorama das tecnologias de orquestração, *schedulers* de HPC, containerização, orquestradores *cloud-native* e plataformas de MLOps;
- Descrever a arquitetura de referência adotada, baseada na *NVIDIA AI Enterprise* (NVAIE) [2];
- Conduzir as POCs em três cenários, sendo eles o HPC tradicional em *bare-metal* (A), o ecossistema *cloud-native* com fatiamento virtual de *Graphics Processing Units* (GPUs) (B) e por fim a arquitetura de referência da NVIDIA (C), registrando, em cada cenário, os passos realizados, os resultados alcançados e a forma de compartilhamento e uso do ambiente disponível;
- Reportar o estado atual da implantação física e lógica do ambiente e definir os próximos passos rumo à disponibilização para a comunidade usuária.

Metodologicamente, as POCs foram executadas em ambiente virtualizado de prototipação e restringiram-se à validação lógica, de orquestração e de submissão de tarefas, sem ensaios de estresse físico do *hardware*. As observações registradas têm natureza qualitativa e apoiam-se em referências da literatura científica e em boas práticas de engenharia; não se pretende, com elas, estabelecer uma medida comparativa de desempenho entre os três cenários. Esse recorte limita

o alcance das conclusões à camada lógica, ou seja, não sustenta afirmações sobre desempenho sob carga nem sobre o comportamento do *hardware* em regime de produção.

O restante do documento organiza-se como segue. A Seção 2 apresenta o panorama tecnológico das ferramentas de orquestração. A Seção 3 detalha a arquitetura de referência adotada, baseada na suíte NVAIE. A Seção 4 descreve as POCs e a análise comparativa das alternativas. A Seção 5 consolida o estado atual da implantação e as frentes de trabalho subsequentes. A Seção 6 apresenta as conclusões e considerações finais.

2 Panorama Tecnológico

A orquestração de recursos em ambientes HPC compartilhados entre múltiplos usuários é um desafio fundamental em centros de computação modernos [3]. Historicamente, sistemas HPC foram gerenciados através de *schedulers* dedicados, ou seja, softwares responsáveis por arbitrar o acesso aos recursos e definir a ordem de execução dos *jobs* [4]. Exemplos notórios nesta categoria incluem o Slurm, um gerenciador de recursos e de carga de trabalho, de código aberto, amplamente adotado em supercomputadores [5], e o PBS Pro, um escalonador robusto focado em ambientes de alto desempenho [6]. Esses sistemas otimizam a utilização de *clusters* por meio de algoritmos sofisticados de alocação de *jobs*, os quais são tarefas computacionais submetidas pelos usuários para processamento em lote [4, 7]. Contudo, a convergência entre computação em nuvem e HPC, impulsionada pelo crescimento exponencial da demanda por recursos de IA, criou a necessidade de plataformas mais flexíveis que possam atender tanto aos fluxos de trabalhos tradicionais quanto aos baseados em contêineres e microsserviços [8].

Para mapear essa transição tecnológica e compreender o ecossistema atual, as subseções seguintes estão organizadas de forma estrutural e progressiva. Inicialmente, abordam-se gerenciadores de tarefas consolidados no ambiente de HPCs tradicionais, Seção 2.1, as tecnologias de *runtimes* e containerização adaptadas para alto desempenho, Seção 2.2. Em seguida, analisa-se o paradigma de orquestradores de contêineres *cloud-native*, Seção 2.3, bem como os escalonadores de tarefas e *middlewares*, de categoria *cloud-native*, construídos para viabilizar sua aplicação em simulações paralelas, Seção 2.4. Na sequência, avançamos para a camada de aplicação, detalhando as plataformas de MLOps e interfaces de usuário, Seção 2.5, concluindo com as ferramentas de orquestração para Aprendizado Federado, Seção 2.6.

2.1 Gerenciadores de Cargas de Trabalho (HPC Tradicional)

Para garantir a execução eficiente de simulações e processamentos em larga escala, os centros de HPC historicamente utilizam, como parte de sua estrutura fundacional, gerenciadores de *jobs* projetados para orquestrar o acesso direto aos recursos físicos (*bare-metal*) do *cluster*. O foco principal desta abordagem é maximizar a utilização de *Central Processing Units* (CPUs), GPUs e tráfego de redes de alta velocidade, através do enfileiramento rigoroso e da execução otimizada de *jobs* em lote (*batch processing*). Para ilustrar este ecossistema, as subseções a seguir descrevem dois dos escalonadores mais consolidados e historicamente relevantes na literatura para a orquestração de infraestruturas tradicionais [9, 5, 6]: o Slurm, apresentado na Seção 2.1.1, e o PBS Pro, apresentado na Seção 2.1.2.

2.1.1 Slurm

O gerenciador de tarefas Slurm é amplamente reconhecido como o estado da arte na alocação eficiente de recursos para HPC [10], destacando-se por maximizar a utilização do *cluster* e minimizar o tempo de espera de *jobs* por meio de algoritmos avançados, como o *backfilling* e o escalonamento baseado em *fair-share*, e predição de demanda [5]. Sua arquitetura distribuída, ilustrada na Figura 2, baseia-se em componentes especializados operando em conjunto. Entre eles, destacam-se:

- **slurmctld** - controlador central responsável pelo agendamento da fila;
- **slurmd** - agente local de execução nos nós de computação;
- **slurmdbd** - gerenciador de banco de dados para bilhetagem, limites e histórico;

- **slurmrestd** - interface de *Application Programming Interface* (API) do tipo *Representational State Transfer* (REST) destinada à comunicação e integração com plataformas externas.

Embora a plataforma seja robusta na alocação de cargas de trabalho heterogêneas, três características de seu modelo de operação a distanciam de paradigmas computacionais emergentes. A primeira delas é que o isolamento de contêineres restringe-se essencialmente a *cgroups*, sem o confinamento de rede e de sistema de arquivos típico dos orquestradores.

Além disso, a integração com esteiras modernas de *Continuous Integration/Continuous Deployment* (CI/CD) depende de *wrappers* sobre a interface de linha de comando, e não de uma API declarativa; e as políticas de *multi-tenancy* são expressas em tempo de configuração, de modo que classes de *Quality of Service* (QoS) granulares, como a atribuição de frações de GPUs a contas distintas, exigem reconfiguração administrativa em vez de provisionamento sob demanda [3, 11].

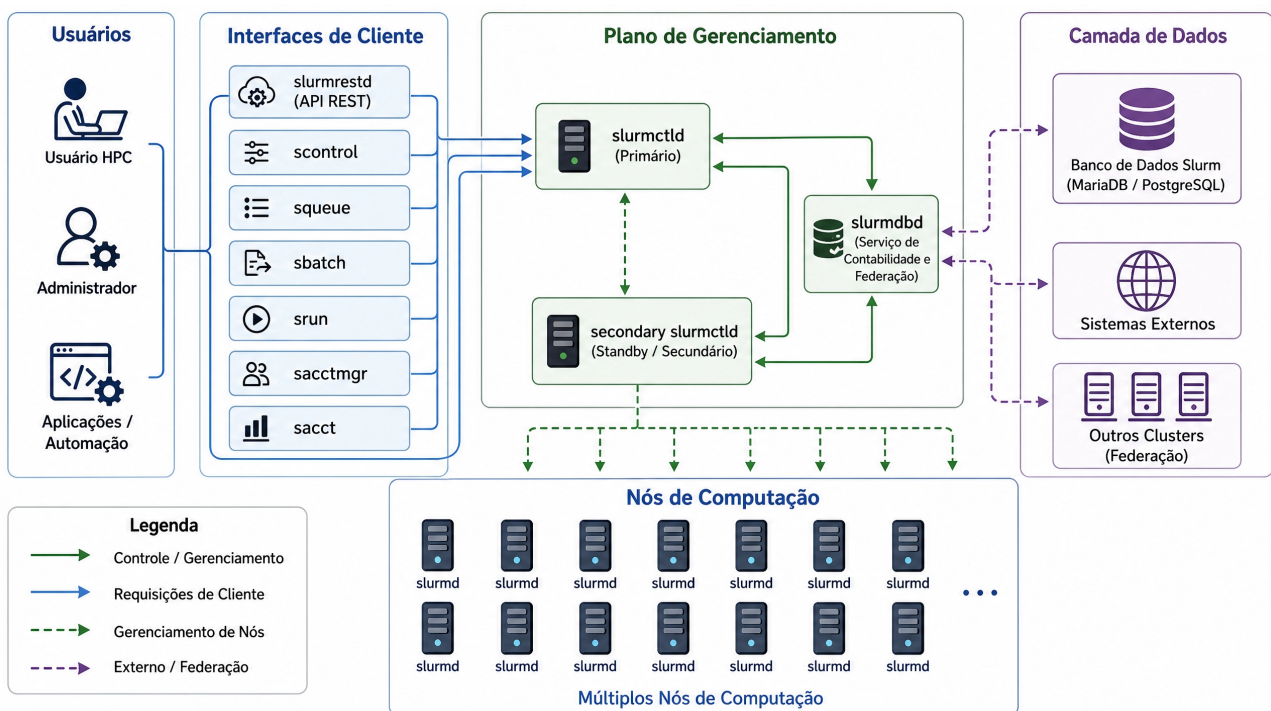


Figura 2: Ilustração dos componentes do *Slurm* (Gerado por IA).

2.1.2 PBS Pro

O *Portable Batch System* (PBS) é reconhecido como uma das arquiteturas mais maduras e escaláveis para o gerenciamento de cargas de trabalho em HPC corporativo. A ferramenta destaca-se por seu motor de agendamento fortemente baseado em políticas, denominado *Policy Driven Scheduling* (PDS), que permite a criação de regras complexas de priorização, preempção de *jobs* e topologia ciente de rede [12]. Embora ofereça extrema robustez para cargas de trabalho monolíticas e alta eficiência no uso de CPU/GPU em simulações em lote, o PBS compartilha das mesmas limitações arquiteturais do Slurm no contexto de convergência para nuvem HPC. A plataforma não foi concebida para o ecossistema *cloud-native*, apresentando atritos na orquestração dinâmica de contêineres, ausência de abstrações para microsserviços

e limitações na gestão elástica de fatiamento virtual de GPU, tornando sua integração com plataformas de MLOps, como o Kubeflow, complexa e não-nativa [13].

2.2 Containerização em HPC

A adoção de tecnologias de containerização em ambientes de HPC impõe desafios que divergem significativamente do ecossistema de nuvem tradicional, exigindo soluções que respeitem o modelo de segurança e a eficiência do *hardware* de alto desempenho. Como as arquiteturas clássicas lidam com múltiplos usuários compartilhando os mesmos recursos físicos, a introdução de contêineres requer ferramentas especializadas que evitem gargalos de rede e vulnerabilidades de acesso. Para mitigar essas restrições operacionais, a Seção 2.2.1 a seguir detalha o Apptainer, evidenciando como a ferramenta se consolidou como o padrão arquitetural para garantir o isolamento e a portabilidade das cargas de trabalho sem comprometer a integridade do *cluster*.

2.2.1 Apptainer (Singularity)

Desenvolvido especificamente para atender às restrições de segurança e arquitetura de supercomputadores, o Apptainer (anteriormente conhecido como Singularity) opera como um *runtime* de contêineres alternativo ao Docker [14]. Em *clusters* compartilhados, o modelo clássico, baseado em Docker, apresenta sérios riscos devido à necessidade de *daemons* executados com privilégios de administrador. O Apptainer resolve essa vulnerabilidade executando os contêineres como processos comuns do usuário, eliminando qualquer possibilidade de escalada de privilégios [15]. Além disso, a ferramenta oferece integração nativa com sistemas de arquivos paralelos distribuídos, variáveis de ambiente locais e rede de alta velocidade (*InfiniBand* ou *RDMA over Converged Ethernet (RoCE)*), tornando-se a solução padrão para garantir a portabilidade e a reprodutibilidade de aplicações de IA sobre escalonadores como o Slurm.

2.3 Orquestradores de Contêineres (*Cloud-Native*)

Em contraste com os gerenciadores de *workloads* tradicionais focados na execução em *bare-metal*, o paradigma *cloud-native* introduziu uma nova abordagem por meio da containerização. Os orquestradores de contêineres foram desenvolvidos fundamentalmente para lidar com arquiteturas baseadas em microsserviços, garantindo isolamento, portabilidade, tolerância a falhas e escalabilidade elástica de forma automatizada [16]. Com a crescente convergência entre as infraestruturas de computação em nuvem e os ecossistemas de HPC, impulsionada fortemente pelas esteiras de desenvolvimento de IA, essas ferramentas tornaram-se vitais para assegurar a reprodutibilidade e a agilidade nos ambientes de pesquisa e produção [17]. A Seção 2.3.1 descreve o principal representante deste ecossistema.

2.3.1 Kubernetes

O K8S é consolidado como o padrão para a orquestração de contêineres em ambientes *cloud-native*, oferecendo uma camada de abstração unificada que simplifica o gerenciamento de recursos, a escalabilidade horizontal e a alta disponibilidade de aplicações [18]. Sua arquitetura, ilustrada na Figura 3, é composta por um *Control Plane* central (abrigoando o *API Server*, o banco de dados *etcd* e o *Kube-scheduler*) e nós de execução (*Worker Nodes*) que operam através do agente *kubelet*.

Componentes do Kubernetes

-  **Node:** Um servidor adicionado a um cluster K8s.
-  **Cluster:** Uma coleção de um ou mais nodes.
-  **Namespace:** Um ambiente isolado que contém objetos nomeados de forma única.
-  **Container:** Uma aplicação auto-contida e pronta para implantação.
-  **Pod:** Um container e seus metadados associados, etc.
-  **Volume:** Armazenamento que pode ser compartilhado no K8s.
-  **Service:** Networking, portas, etc.
-  **Job:** Um Job implanta um Pod que executa uma "tarefa" e finaliza.
-  **DaemonSet:** Um objeto que requer uma réplica de um Pod rodando em cada node.
-  **Deployment:** Um objeto que garante que um Pod esteja rodando e reimplanta uma cópia se ele for deletado.

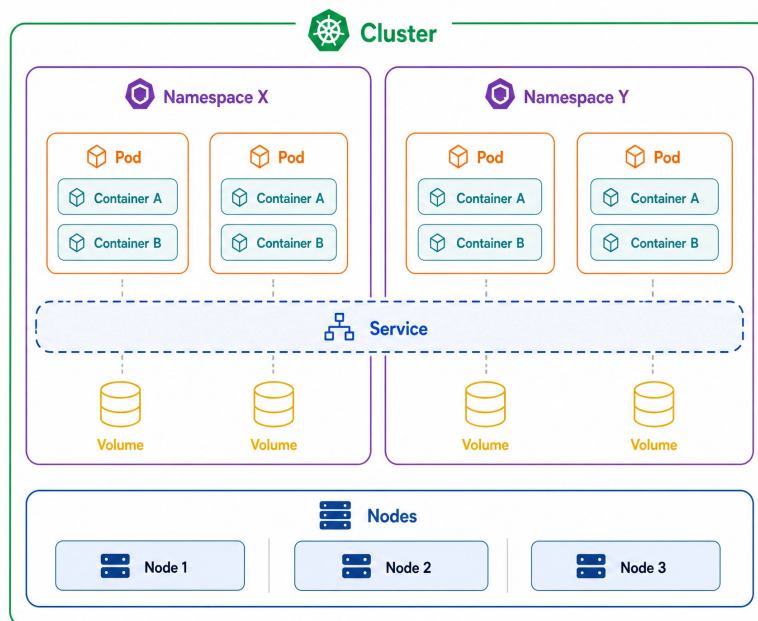


Figura 3: Ilustração dos componentes do K8S (Gerado por IA).

Para cargas de trabalho de IA, o ecossistema destaca-se pelo uso de operadores, como o *NVIDIA GPU Operator*, que automatiza o provisionamento de *drivers*, *runtime* de contêineres e o monitoramento [19]. Embora seja a plataforma ideal para o *deploy* de microsserviços e *pipelines* de MLOps, o Kubernetes apresenta limitações em cenários de HPC puro, pois seu agendador de tarefas nativo foi projetado para tarefas de longa duração e não possui suporte a estratégias como *gang-scheduling*, o que resulta em subutilização de recursos e latências elevadas em *jobs* de categoria *Message Passing Interface* (MPI), ou seja, massivamente paralelos [11].

2.4 Escalonadores Avançados e *Middleware* para Kubernetes

Conforme evidenciado, embora o K8S ofereça uma infraestrutura resiliente e elástica para microsserviços, seu escalonador padrão apresenta lacunas significativas para a execução eficiente de *workloads* de HPC e IA [20]. Para transpor essas barreiras e viabilizar a convergência definitiva para um ambiente *cloud-native* em HPC e IA, emergiu um ecossistema de *schedulers* customizados e *middlewares* especializados [13]. Essas ferramentas estendem as capacidades nativas do orquestrador ao introduzir primitivas clássicas de HPC diretamente no nível do contêiner, como *gang-scheduling*, gerenciamento de filas hierárquicas e o fatiamento dinâmico de GPUs. As Subseções 2.4.1, 2.4.2 e 2.4.3 detalham, respectivamente, três soluções proeminentes que adotam diferentes abordagens para este desafio: o *Volcano*, focado em *batch-scheduling*; o *Slinky*, que propõe uma unificação ao embutir o motor do *Slurm* nativamente no Kubernetes; e o *Run:ai*, uma plataforma corporativa dedicada à otimização e orquestração de GPUs para *pipelines* de MLOps.

2.4.1 Volcano

Para mitigar as limitações do *scheduler* nativo do K8S em cenários de alta concorrência e dependência paralela, o Volcano apresenta-se como um sistema de *batch-scheduling cloud-native* avançado, incubado pela *Cloud Native Computing Foundation* (CNCF) [21]. Diferente da

abordagem orientada a serviços de longa duração do Kubernetes, o motor do Volcano introduz primitivas clássicas de HPC no ecossistema de contêineres, como suporte a múltiplas hierarquias de filas, *fair-share scheduling* e gerenciamento rigoroso de cotas. Seu maior diferencial técnico para cargas de IA distribuída é a implementação nativa de *gang-scheduling* garantindo que um conjunto de *pods* inter-relacionados seja alocado simultaneamente, prevenindo interrupções (*deadlocks*) e fragmentação de recursos de GPU [21, 13]. Na arquitetura avaliada, o Volcano atuou como a inteligência de orquestração sobre o Kubernetes, fornecendo a base necessária para a camada de fatiamento virtual de GPUs.

2.4.2 Slinky

Representando uma abordagem direta de convergência arquitetural, o projeto Slinky visa unificar os paradigmas de HPC e *cloud-native* ao executar o ecossistema Slurm nativamente sobre um *cluster* Kubernetes. Em vez de operar os orquestradores em silos separados ou através de pontes externas, o Slinky utiliza o padrão de operadores do Kubernetes e *Custom Resource Definitions* (CRDs) para provisionar e gerenciar o ciclo de vida dos módulos do Slurm (como `slurmctld` e `slurmd`) operando-os como *pods* [22]. Essa topologia permite que os pesquisadores continuem submetendo *jobs* por meio das interfaces tradicionais (como `sbatch` ou `srun`), enquanto a infraestrutura beneficia-se da elasticidade do Kubernetes. Contudo, a abstração introduz uma nova camada de complexidade operacional. A estabilidade do ambiente híbrido exige um controle rigoroso sobre o estado dos *pods* e a gestão síncrona de armazenamento distribuído através de *Persistent Volume Claims* (PVCs), elementos que demandam monitoramento constante e podem ampliar a complexidade da gestão.

2.4.3 Run:ai

Posicionado como uma plataforma corporativa especializada na orquestração de cargas de trabalho de IA, o Run:ai opera sobre *clusters* Kubernetes para preencher a lacuna entre a infraestrutura bruta e as demandas dinâmicas de MLOps. O núcleo da plataforma é um *scheduler* customizado que substitui a lógica padrão do Kubernetes, introduzindo algoritmos avançados de *fair-share*, gerenciamento hierárquico de cota e alocação de recursos [23].

O principal diferencial técnico da arquitetura, ilustrada na Figura 4, é a sua tecnologia de fatiamento dinâmico, que permite a multiplexação transparente da memória e GPU entre múltiplos contêineres. Ao contrário de segmentações estáticas baseadas em *hardware* (*Multi-Instance GPU* (MIG)), a abstração em nível de *software* do Run:ai maximiza a utilização do *cluster*, permitindo alta densidade de ambientes interativos (como instâncias do *JupyterHub*) sem sacrificar a capacidade da infraestrutura de priorizar e orquestrar nativamente o treinamento de modelos distribuídos em larga escala.

2.5 Plataformas de MLOps e Interfaces de Usuário

Enquanto as camadas de orquestração e escalonamento abordadas nas seções anteriores focam na alocação eficiente e no fatiamento de recursos físicos, a adoção prática de um ambiente de HPC voltado para IA depende diretamente da usabilidade fornecida aos cientistas de dados e pesquisadores [24]. Para isolar os usuários da complexidade intrínseca da infraestrutura subjacente, como a sintaxe de submissão de *jobs* e as configurações de rede, implementa-se uma camada de plataformas MLOps e interfaces de usuário. Estas ferramentas abstraem o provisionamento, gerenciando todo o ciclo de vida do modelo: desde a prototipagem interativa

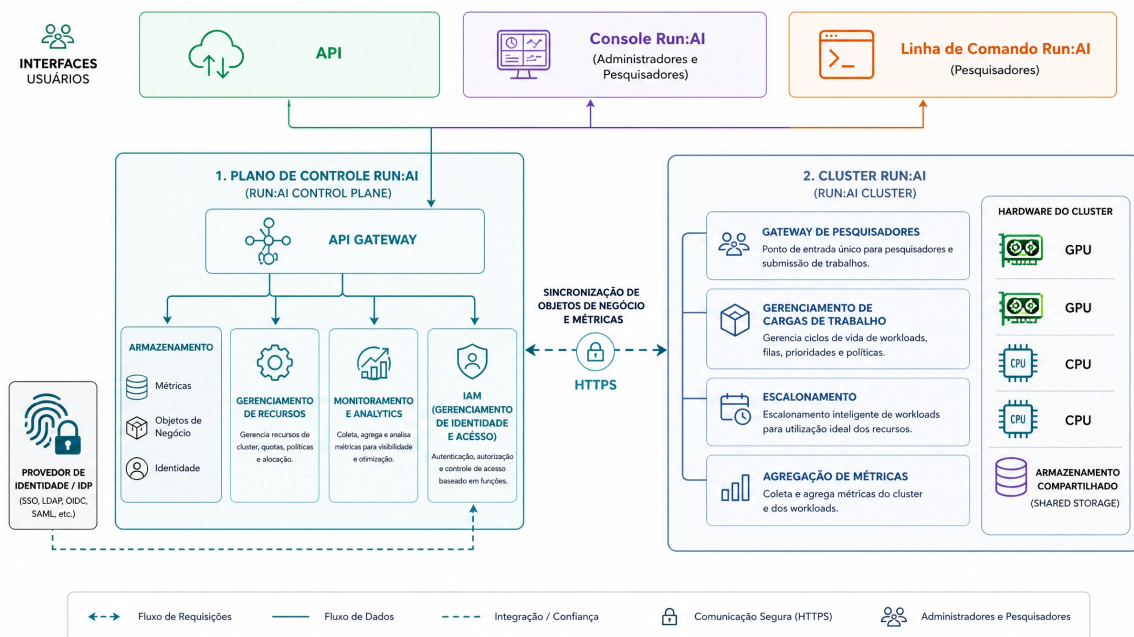


Figura 4: Ilustração dos componentes do Run:ai (Gerado por IA).

de código até a produtização e automação de *pipelines* reproduzíveis [25]. As subseções a seguir detalham três soluções que operam neste nível de abstração: na Seção 2.5.1 temos o JupyterHub, consolidado como interface interativa padronizada; na Seção 2.5.2 temos o KubeFlow, projetado para orquestrar fluxos de trabalho avançados de aprendizado de máquina sobre Kubernetes; e por fim na Seção 2.5.3 temos o Almoda, voltado para a rápida construção de ambientes de experimentação isolados.

2.5.1 JupyterHub

Atuando como a principal interface interativa para pesquisadores, o JupyterHub orquestra o provisionamento de ambientes de desenvolvimento baseados em *notebooks* para múltiplos usuários de forma isolada. Diferente de um gerenciador de *cluster* puro, sua arquitetura atua na camada de aplicação, delegando a alocação real dos recursos computacionais para o orquestrador subjacente através de mecanismos de *spawning* como o *KubeSpawner* em ambientes Kubernetes ou o *BatchSpawner* integrado ao Slurm [26]. Nos cenários avaliados, a plataforma demonstrou ser a ferramenta ideal para exploração de dados, depuração interativa e prototipagem ágil de modelos de IA. O JupyterHub atua como uma ponte transparente, permitindo que os usuários consumam o poder computacional e o fatiamento de GPUs do *cluster* sem precisar dominar a complexidade das linhas de comando da infraestrutura.

2.5.2 KubeFlow

Concebido como uma plataforma nativa do Kubernetes para MLOps, o KubeFlow atua na orquestração do ciclo de vida lógico dos modelos de IA. Enquanto as ferramentas da camada inferior gerenciam o *hardware*, o *KubeFlow* provê abstrações padronizadas para treinamento de modelos, execução de experimentos, ajuste distribuído de hiperparâmetros (*Katib*) e, fundamentalmente, a orquestração de *pipelines* reproduzíveis e escaláveis [27]. A avaliação da ferramenta

confirmou sua capacidade vital de transformar experimentos iterativos, inicialmente criados no JupyterHub, em esteiras de produção automatizadas. Contudo, a estabilidade e o desempenho dessas esteiras são estritamente dependentes da capacidade do *scheduler* subjacente (como Volcano) em alocar, enfileirar e fatiar as GPU exigidas por cada etapa do fluxo de trabalho.

2.5.3 Alnoda

Diferente de ecossistemas corporativos robustos voltados para a governança de múltiplos usuários concorrentes, o Alnoda apresenta-se como uma plataforma focada na criação de ambientes de desenvolvimento ágeis e isolamento baseado em contêineres [28]. A ferramenta empacota em imagens padronizadas utilitários fundamentais para o cientista de dados, incluindo *Integrated Development Environments* (IDEs) baseadas em *web browser*, terminais, gerenciadores de arquivos e ferramentas de visualização gráfica. Embora não possua os mecanismos de escalonamento multi-nó de plataformas como o KubeFlow, o Alnoda destaca-se em cenários de prototipagem rápida e validação local, servindo como uma solução eficiente para simular o comportamento de aplicações de IA e monitorar de forma controlada a competição por recursos de aceleração gráfica em ambientes de escala reduzida.

2.6 Orquestração de Aprendizado Federado

À medida que o desenvolvimento de modelos de IA avança para além dos limites físicos de um único *cluster* centralizado, o Aprendizado Federado consolida-se como um paradigma arquitetural essencial. Essa abordagem viabiliza o treinamento colaborativo através de múltiplas instituições ou dispositivos de borda (*edge*), mantendo os dados brutos estritamente locais para garantir a conformidade regulatória e a privacidade [29]. Contudo, coordenar essa carga de trabalho altamente distribuída exige plataformas capazes de gerenciar a sincronização global, a comunicação de redes muitas vezes instáveis e a agregação matemática dos pesos dos modelos. Para solucionar esses desafios operacionais, emergiram orquestradores dedicados a este ecossistema [30]. As subseções a seguir exploram duas das soluções mais proeminentes desta área, que operam de forma complementar: a Seção 2.6.1 apresenta o *Flower*, focado na agilidade e flexibilidade algorítmica para pesquisa, enquanto na Seção 2.6.2 temos o NVIDIA *Federated Learning Application Runtime Environment* (FLARE), voltado para a governança estrutural e segurança em implantações de nível corporativo.

2.6.1 Flower

Posicionado como um *framework* de código aberto e agnóstico a ferramentas de *Machine Learning* (ML) (suportando nativamente as bibliotecas PyTorch, TensorFlow e JAX), o *Flower* consolidou-se como o padrão acadêmico para pesquisa e prototipagem em Aprendizado Federado. Sua arquitetura abstrai a complexidade das comunicações distribuídas, oferecendo uma interface limpa e uma vasta biblioteca de estratégias de agregação (como FedAvg e FedProx) [31]. Embora ofereça extrema flexibilidade para o desenvolvimento lógico e algorítmico, o Flower, em sua concepção base, frequentemente delega as garantias estritas de segurança de tráfego, autenticação e tolerância a falhas em larga escala para a infraestrutura de rede subjacente.

2.6.2 NVIDIA FLARE

Enquanto o Flower destaca-se na pesquisa algorítmica, o NVIDIA FLARE atua na camada de infraestrutura e governança. A plataforma da NVIDIA foca na segurança da topologia federada, oferecendo provisionamento seguro de clientes, comunicação criptografada, auditoria e resiliência contra falhas de conexão [32]. Soluções avançadas no estado da arte têm explorado a integração entre ambas as tecnologias, encapsulando a lógica de agregação do *Flower* sobre o motor de orquestração seguro e corporativo do NVIDIA FLARE.

3 Arquitetura Proposta: *Suíte NVIDIA AI Enterprise*

Após a fundamentação teórica sobre os paradigmas de orquestração em HPC e ambientes *cloud-native*, esta seção detalha a arquitetura de referência adotada para o projeto. A infraestrutura baseia-se no NVAIE, uma suíte de *softwares* certificada e otimizada para o desenvolvimento, implantação e escalonamento de cargas de trabalho de IA e HPC [2]. O NVAIE não atua como uma ferramenta isolada, mas sim como uma pilha tecnológica construída em camadas interdependentes, como ilustra a Figura 5 [2], projetada para extrair o desempenho máximo do *hardware* acelerado, mitigando as complexidades de integração entre ferramentas.

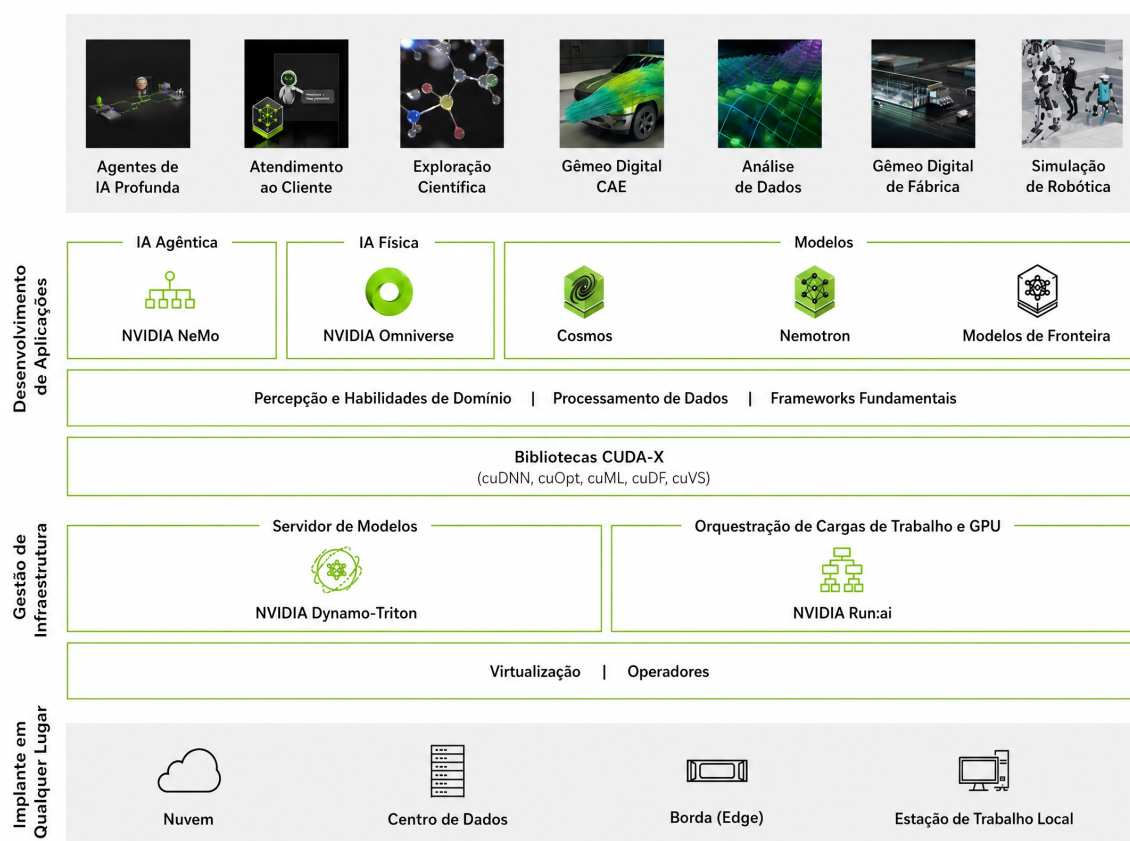


Figura 5: Ilustração da solução NVAIE.

A apresentação da suíte acompanha, nas seções seguintes, a mesma estratificação ilustrada na Figura 5, da infraestrutura física à interface de autoatendimento. A Seção 3.1 descreve o *Base Command Manager* (BCM), responsável pelo provisionamento e pelo gerenciamento centralizado dos nós do *cluster*. A Seção 3.2 mostra como esse mesmo domínio de gerência se estende geograficamente para as regiões de borda. A Seção 3.3 trata da camada de orquestração de contêineres, na qual o NVIDIA GPU Operator torna os aceleradores consumíveis pelo Kubernetes. A Seção 3.4 apresenta as bibliotecas e os *frameworks* otimizados que a suíte disponibiliza aos usuários finais. Por fim, a Seção 3.5 detalha a camada de MLOps, na qual a plataforma Run:ai concentra a alocação de recursos e o acesso dos pesquisadores ao ambiente.

3.1 Provisionamento e Gerenciamento com o BCM

A arquitetura proposta é implementada através do BCM, o qual atua como o sistema operacional do *cluster*, abstraindo a complexidade de provisionamento de nós heterogêneos [33]. Por meio de interfaces automatizadas, a plataforma gerencia de forma centralizada a instalação de imagens do sistema operacional, o mapeamento de sistemas de arquivos distribuídos e a consistência de *firmwares*, possibilitando a administração do ecossistema por meio de uma interface gráfica, como mostrado na Figura 6.

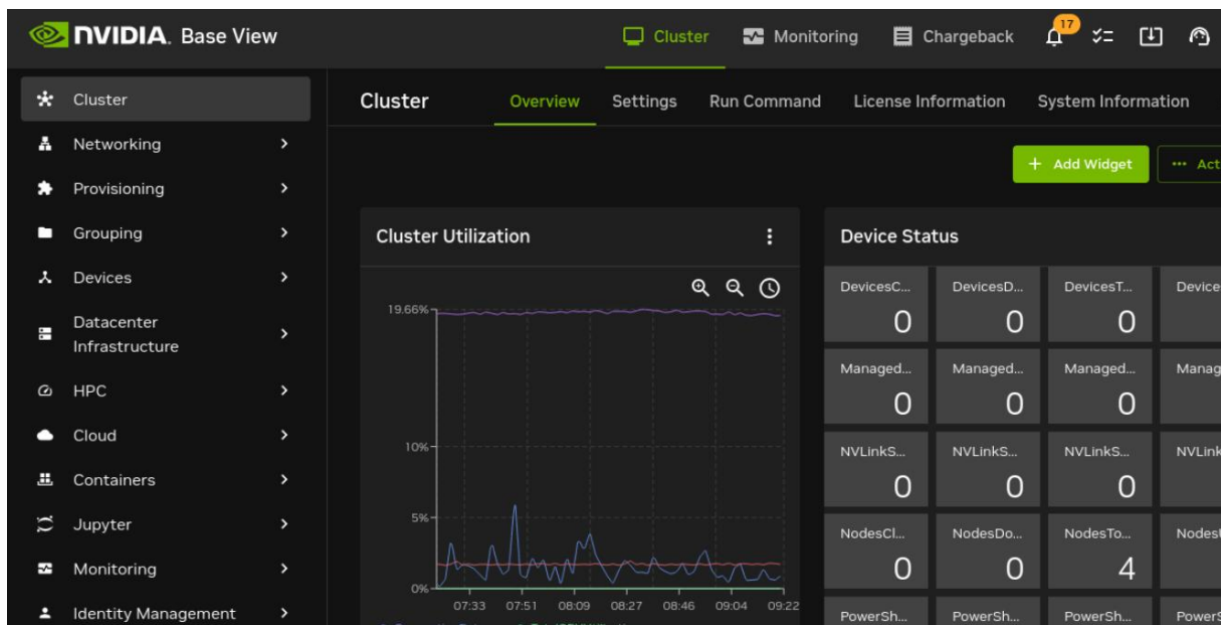


Figura 6: Interface *web* do gestor BCM.

Através deste portal, é possível realizar o controle granular de dispositivos e o agrupamento lógico de nós de computação, conforme perfis específicos de *hardware*, projeto ou arquitetura. A grande agilidade operacional da ferramenta reside em seus assistentes automatizados de implantação, destacando-se pela interface gráfica que permite a criação do ambiente HPC, o ambiente K8S e o *deploy* do JupyterHub, todos integrados. Essa estrutura é amparada por um gerenciador de identidades que automatiza a sincronização de usuários e políticas de acesso, desde o nível do sistema operacional até as ferramentas de MLOps de mais alto nível, permitindo a gestão de segurança e acesso ao ambiente.

3.2 Orquestração e Extensão para a Borda

Para atender aos requisitos de baixa latência, processamento local e preservação de largura de banda, a arquitetura proposta tem capacidade de estender-se para além dos limites do *data center* principal [1]. Através do recurso de *Edge Computing* do BCM, torna-se possível operar sob o paradigma de um único *cluster* distribuído em múltiplas localizações geográficas [33], como ilustra a Figura 7.

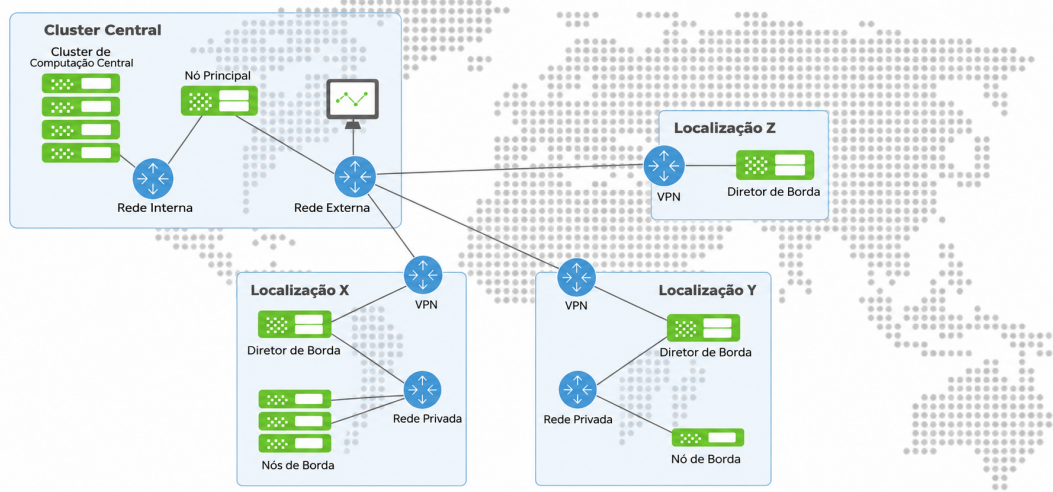


Figura 7: Estrutura geograficamente distribuída.

Esta abordagem concede autonomia aos dispositivos remotos, mitigando a dependência contínua de conectividade com o nó principal [33]. A topologia de extensão para a borda baseia-se na divisão de responsabilidades entre duas entidades lógicas fundamentais operadas nativamente pelo BCM:

- *Edge Directors*: atuam como os coordenadores regionais da infraestrutura de borda e devem ser instalados geograficamente próximos aos nós de computação locais. O *Edge Director* é provisionado inicialmente a partir do *head node*, através de uma imagem de instalação gerada pela ferramenta *cm-edge-setup*. Uma vez em estado ativo, o diretor assume total autonomia para operações locais críticas, como por exemplo:
 - Criação de sistemas de arquivos em memória para os nós locais;
 - Controle de energia do seu próprio *hardware* e dos nós subordinados;
 - Verificação ativa do estado dos dispositivos via requisições baseadas em *Internet Control Message Protocol* (ICMP);
 - Monitoramento de saúde e telemetria fina da infraestrutura de borda.

O *Edge Director* atua também como uma ponte de comunicação, encaminhando pedidos para o *head node* central apenas quando estritamente necessário.

- *Edge Nodes*: são os nós de execução de tarefas localizados na borda da rede. Ao contrário dos nós convencionais do *cluster*, os *Edge Nodes* não necessitam de qualquer ligação direta ao *head node* central, pois eles realizam o processo de arranque e provisionamento via rede diretamente a partir do seu *Edge Director* local.

O grande diferencial desta arquitetura para o projeto é a capacidade de executar localmente sistemas de gestão de cargas de trabalho. O *Workload Manager* (WLM) pode ser estendido nativamente para as regiões de borda. Esta extensão permite que os treinamentos massivos centralizados e a gestão global sejam controlados no *cluster* principal, enquanto as tarefas de alta velocidade baseadas em sensores de borda e os clientes locais de aprendizado federado sejam orquestrados autonomamente nas bordas [33, 34].

3.3 Infraestrutura CNCF: NVIDIA GPU Operator

Para que a camada de orquestração baseada em Kubernetes consiga consumir de forma nativa e segura os recursos de aceleração de *hardware* estruturados pelo BCM, a arquitetura incorpora o NVIDIA GPU Operator e o NVIDIA Network Operator. O uso de operadores estende a API do Kubernetes através de CRD, automatizando o provisionamento de todos os componentes necessários para habilitar o uso de GPU em contêineres [19].

O ciclo de injeção automatizado do NVIDIA GPU Operator engloba o gerenciamento dos seguintes módulos críticos nos nós de trabalho:

- *NVIDIA Container Toolkit*: garante que o *runtime* de contêineres (como o Docker ou *containerd*) consiga expor as GPUs para dentro dos *pods*;
- *Device Plugin* e *Node Feature Discovery* (NFD): mapeiam e rotulam de forma dinâmica as capacidades do *hardware* (quantidade de memória, modelo da GPU e suporte a MIG), permitindo que o *Kube-scheduler* ou o Volcano tomem decisões assertivas de alocação.
- *Data Center GPU Manager* (DCGM) *Exporter*: integra o DCGM ao ecossistema de monitoramento, expondo métricas de telemetria para o monitoramento.

Em paralelo, os *Network Operators* configuram os *drivers* de rede, permitindo o isolamento de canais de comunicação e viabilizando o uso dos recursos de *Remote Direct Memory Access* (RDMA) e NVLink, uma tecnologia proprietária de interconexão que provê comunicação direta e de alta largura de banda entre aceleradores [35], reduzindo a latência de comunicação entre GPUs.

3.4 NVAIE e Bibliotecas Otimizadas

Em relação à suíte de ferramentas do NVAIE, um dos papéis primordiais é fornecer estabilidade, suporte corporativo e aceleração para os *frameworks* de aprendizado de máquina consumidos pelos usuários finais [2]. O acesso a essa camada ocorre majoritariamente através do catálogo *NVIDIA GPU Cloud* (NGC), o qual distribui contêineres pré-compilados e validados para arquiteturas de GPUs modernas. Os três pilares de *softwares* são:

- *Frameworks* de Treinamento Otimizados: imagens personalizadas de PyTorch e TensorFlow que incluem otimizações.
- RAPIDS: uma suíte de bibliotecas que move todo o *pipeline* de ciência de dados (manipulação de *dataframes* e algoritmos com as bibliotecas cuDF e cuML) para dentro das GPUs, eliminando gargalos de transferência de memória entre CPU e GPU.
- NVIDIA Triton e TensorRT: ferramentas dedicadas à otimização e compilação de modelos para produção. O TensorRT realiza a fusão de camadas de redes neurais e quantização de precisão (redução de 32 *bits* para 16 ou 8 *bits*), enquanto o Triton gerencia o ciclo de vida da inferência com suporte a múltiplos *backends*.

3.5 Integração da Camada de MLOps no Ecossistema

A plataforma Run:ai desempenha um papel central como a interface primária para engenheiros de IA e pesquisadores, consolidando a coesão do ecossistema através de uma camada de gerenciamento de alto nível. Conforme ilustrado na Figura 8 [2], o sistema oferece uma estrutura robusta que integra o ciclo de vida completo da IA, permitindo que as equipes construam, treinem e sirvam modelos de maneira fluida dentro de uma arquitetura aberta.

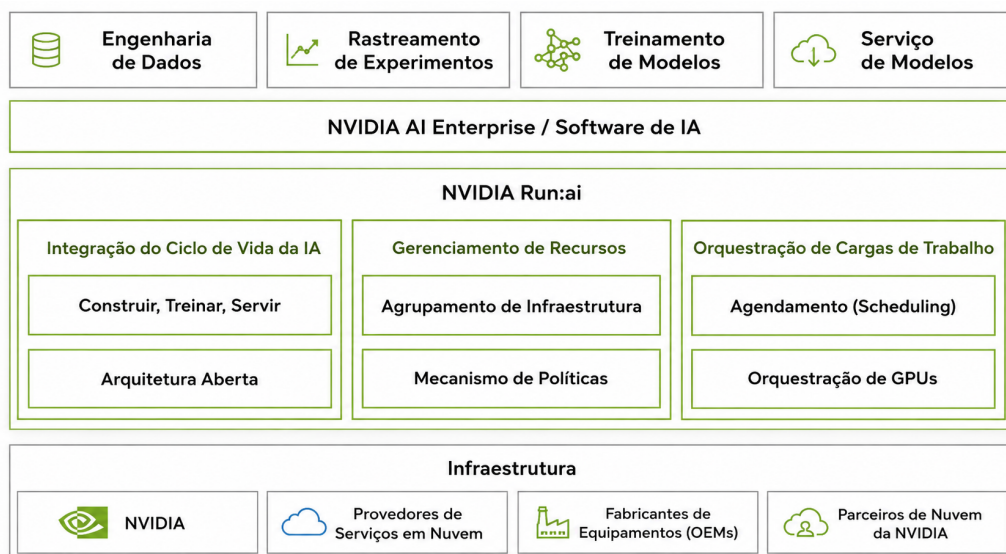


Figura 8: Arquitetura integrada do Run:ai.

Um dos diferenciais fundamentais do Run:ai é atuar na orquestração e gerenciamento inteligente de recursos computacionais. Em vez de depender de intermediários externos para lidar com a complexidade da infraestrutura, a plataforma disponibiliza um painel unificado que facilita a gestão de cargas de trabalho em diferentes ambientes, centralizando o controle e a visibilidade.

Além disso, o Run:ai otimiza a eficiência operacional mediante um mecanismo de políticas avançado, que inclui o agrupamento de infraestrutura e a orquestração eficiente de GPUs. Dentro dessa configuração, os ativos organizacionais são distribuídos de forma estratégica entre projetos e departamentos, permitindo a definição de cotas de uso e a aplicação de flexibilidade preemptiva, o que garante que a alocação de recursos responda dinamicamente às demandas de uso corrente da organização.

3.5.1 Fluxo de Trabalho Interativo

O pesquisador acessa diretamente o catálogo de ferramentas do Run:ai para instanciar e provisionar o ambiente ideal para seu código, sem a necessidade de interagir manualmente com os manifestos do Kubernetes ou com os comandos de submissão do Slurm subjacentes. O fluxo pode ser baseado na fase de treinamento ou inferência [2] e está ilustrado nas Figuras 9 e 10.

No que diz respeito ao treinamento, o fluxo no Run:ai é otimizado para lidar com modelos de longa duração que não admitem interrupções, exigindo apenas a imagem de contêiner com o código e os *datasets* correspondentes. O sistema é projetado para maximizar a utilização de recursos através de um *design* sujeito à preempção, o que permite que os treinamentos operem com cotas flexíveis, podendo até exceder os limites padrão caso haja disponibilidade no *cluster*. Essa flexibilidade é particularmente vantajosa em ambientes multiusuário, nos quais os períodos de ociosidade de outros projetos podem ser aproveitados de forma oportunista, elevando a taxa média de utilização das GPUs do *cluster* como um todo.

Já no fluxo de inferência, o foco central é garantir a continuidade do serviço e a alta disponibilidade exigida em ambientes de produção. Diferente do treinamento, as cargas de inferência não estão sujeitas à preempção, operam estritamente dentro das cotas atribuídas e contam com recursos de autoescalonamento dinâmico baseados em métricas como concorrência e taxa

Treinamento



Figura 9: Fluxo de treinamento.

de requisições, garantindo prioridade e baixa latência. Essa distinção reflete a natureza complementar dos dois perfis de carga: enquanto o treinamento tolera variações momentâneas de desempenho em troca de maior aproveitamento de recursos ociosos, a inferência prioriza previsibilidade e tempo de resposta consistente, mesmo que isso implique reserva dedicada de capacidade computacional.

Inferência



Figura 10: Fluxo de inferência.

De forma geral, podemos destacar as seguintes etapas:

1. Seleção do Projeto: vinculação da carga de trabalho a um projeto ou departamento específico, herdando automaticamente as políticas de cotas e prioridades;
2. Associação de Credenciais: injeção segura de chaves de autenticação e segredos de acesso a repositórios ou registros de contêineres;

3. Vínculo de Fontes de Dados: mapeamento automatizado de volumes de armazenamento ou *buckets* de dados necessários para a sessão;
4. Inicialização do Ambiente Interativo: criação do contêiner executando a ferramenta selecionada pelo usuário, como instâncias isoladas de Jupyter, consumindo frações dinâmicas de GPU;
5. Monitoramento Contínuo: acompanhamento em tempo real do consumo de recursos e do estado da sessão através dos painéis nativos do Run:ai, permitindo intervenção rápida em caso de anomalias;
6. Limpeza do Ambiente: desalocação imediata e automática dos recursos do *cluster* assim que a sessão interativa é encerrada pelo usuário, mitigando a ociosidade de *hardware*.

Toda essa orquestração lógica de ativos e ferramentas realizada na interface do Run:ai consome diretamente os operadores e as bibliotecas otimizadas da suíte NVAIE, que por sua vez rodam sobre a infraestrutura física e de rede provisionada na base pelo BCM e K8S. Essa estratificação em camadas bem definidas — interface de autoatendimento, operadores especializados e infraestrutura física — é o que permite ao pesquisador concentrar-se exclusivamente na lógica experimental do seu modelo, delegando por completo os aspectos de provisionamento, segurança e escalonamento à plataforma subjacente. Essa separação de responsabilidades também facilita a manutenção e a evolução independente de cada camada, sem impactar diretamente a experiência do usuário final.

4 POC e Análise de Alternativas

A implantação de uma infraestrutura de HPC exige uma validação rigorosa e antecipada para mitigar riscos operacionais, reduzir o tempo de entrada em produção e garantir que as escolhas arquiteturais atendam plenamente aos casos de uso do projeto [36, 37]. Neste contexto, foram estruturadas POCs focadas estritamente na camada de *software*.

O objetivo central desta POC foi realizar uma avaliação prática e comparativa do ecossistema de *software* proposto. De um lado, busca-se homologar e extrair métricas de usabilidade da suíte proprietária NVAIE, validando suas promessas de automação, segurança e otimização para modelos de IA. Por outro lado, a POC também conduz um estudo prático das ferramentas de código aberto, avaliando a viabilidade, a curva de aprendizado e o esforço de sustentação necessários para manter um ambiente equivalente sem o uso de licenciamento corporativo.

Ao antecipar a instalação definitiva por meio de ambientes virtualizados, a equipe técnica adquiriu o conhecimento prático necessário, identificando precocemente eventuais gargalos de integração e mapeamento de dificuldades.

4.1 Arquitetura e Escopo

As POCs restringem-se à validação lógica, de orquestração e de submissão de *Jobs*, não contemplando testes de estresse de *hardware* (térmicos, elétricos ou de vazão máxima de rede física), uma vez que os ensaios são executados em um ambiente de prototipagem isolado. A arquitetura da POC foi desenhada para garantir máxima agilidade e reprodutibilidade. Para isso, utilizou-se um ambiente de virtualização. O escopo de validação foi dividido em duas frentes tecnológicas principais:

- Ferramenta do pacote NVAIE (Proprietária): avaliação do provisionamento e gerenciamento centralizado de nós utilizando o BCM, em conjunto com o NVIDIA GPU *Operator* e configuração do K8S, JupyterHub e ambiente de borda;
- Ferramentas de código aberto: avaliação de um ecossistema construído sobre ferramentas de código aberto. O escopo inclui a implantação de *clusters* Kubernetes puros e a utilização de operadores da comunidade para o gerenciamento das GPUs. Um foco especial neste escopo é a avaliação do projeto Slinky (Seção 4.3.8), que visa unir a gestão tradicional de filas de HPC à flexibilidade do fatiamento de GPUs e da orquestração de contêineres.

4.2 Cenário A: HPC Tradicional

Como ponto de partida, o Cenário A estabelece a arquitetura clássica e fundamental de ambientes HPC. O objetivo principal desta etapa foi criar uma linha de base para mapear o esforço operacional, grau de dificuldade e flexibilidade exigidos em uma implantação *bare-metal*, servindo como referência comparativa para os cenários mais modernos abordados nas seções subsequentes deste relatório. Essa linha de base é particularmente valiosa por refletir o modelo de operação historicamente consolidado em centros de supercomputação, contra o qual arquiteturas emergentes precisam demonstrar ganhos reais de flexibilidade e produtividade. Para estruturar este ambiente de forma funcional, a implantação metodológica foi construída em camadas e dividida nas etapas a seguir.

Inicialmente, estabelece-se a fundação de *softwares* com a configuração do sistema operacional, do *kernel* e dos *drivers* de GPU, conforme detalhado na Seção 4.2.1. Em seguida,

implementa-se o serviço *MUNGE Uid 'N' Gid Emporium* (MUNGE), detalhado na Seção 4.2.2, requisito indispensável para garantir a confiança e a autenticação criptografada entre as máquinas. Com a malha de segurança operante, a arquitetura do escalonador de tarefas é instanciada através do Slurm, abordando primeiramente o nó controlador central, detalhado na Seção 4.2.3, e posteriormente os nós de processamento como descrito na Seção 4.2.4. Por fim, para viabilizar a execução segura e mitigar conflitos de dependências, a Seção 4.2.5 detalha o isolamento do espaço de usuário por meio do sistema de módulos e da containerização estrita para HPC provida pelo Apptainer, também conhecido como Singularity, que é uma plataforma de contêineres amplamente utilizada em ambientes científicos e de HPC, refletindo décadas de maturidade e confiabilidade consolidadas pela comunidade de computação científica [15].

4.2.1 Fundação de *Softwares* e Aceleração por GPU

Nesta etapa inicial da implantação, estabelece-se a fundação de *software* sobre a qual toda a pilha de orquestração e processamento científico irá operar. A estabilidade e a performance do *cluster* dependem da correta aderência entre o sistema operacional, a versão do *kernel* e os *drivers* proprietários de aceleração de *hardware*.

Para os nós de gerência e de processamento, padronizou-se a utilização do Ubuntu Server 24.04 LTS. A escolha por esta distribuição baseia-se no seu suporte estendido de longo prazo e na ampla compatibilidade nativa com os principais *frameworks* de IA e pacotes de HPC. A instalação foi realizada no modo padrão sem interface gráfica, para reduzir a sobrecarga de pacotes e serviços desnecessários, preservando recursos de CPU e memória estritamente para o processamento de cargas de trabalho. Para suportar a compilação cruzada dos módulos proprietários da NVIDIA, é imprescindível a instalação das dependências de desenvolvimento, especificamente os pacotes `build-essential` e `linux-headers` correspondentes à versão do *kernel* em execução. Após a instalação e reinicialização dos nós, a configuração é validada por meio do utilitário `nvidia-smi`, confirmando a versão do *driver* instalado e o reconhecimento das placas de vídeo.

Neste ponto do processo, destacou-se a importância de verificar o limite de compatibilidade da biblioteca *Compute Unified Device Architecture* (CUDA) com a ramificação do *driver* instalado. Esta validação é fundamental para prevenir falhas de execução em cargas de trabalho que exijam versões específicas das ferramentas de desenvolvimento CUDA, uma vez que *frameworks* como PyTorch e TensorFlow costumam depender de faixas de versão bem definidas do *toolkit* para funcionar corretamente.

4.2.2 Malha de Confiança e Autenticação

A fundação de segurança e confiança em um ambiente de orquestração distribuída como o Slurm depende de um mecanismo robusto para a criação e validação de credenciais. Para este fim, o *cluster* utiliza o serviço MUNGE, o qual permite que processos originados em um nó sejam autenticados nos demais nós do *cluster* sem a necessidade de senhas ou interação do usuário, garantindo que o `slurmctld` e os *daemons* `slurmd` comuniquem-se de forma segura e criptografada, mesmo em redes internas não segmentadas [38].

A consolidação deste mecanismo exige um procedimento de configuração criterioso para garantir a integridade da malha de comunicação. Dessa forma, as subseções subsequentes detalham o fluxo operacional completo para o estabelecimento do serviço, englobando desde a sincronização prévia de usuários e instalação, na Seção 4.2.2.1, passando pela geração e distribuição segura das chaves criptográficas a partir do nó mestre, na Seção 4.2.2.2, pela aplicação

das permissões sistêmicas estritas, na Seção 4.2.2.3, finalizando com a validação funcional do serviço, na Seção 4.2.2.4.

4.2.2.1 Sincronização de Usuários e Instalação

Para o correto funcionamento do MUNGE, requisito este para a instalação do Slurm, é necessária a padronização das identidades de sistema. Em ambientes que gerenciam contas locais nos servidores, é fundamental garantir que os usuários possuam os mesmos identificadores (*User Identifier* (UID) e *Group Identifier* (GID)) em todo o *cluster* antes da instalação. Esta ação é desnecessária em infraestruturas com sistemas centralizados de identidade. Após confirmar essa consistência em todos os nós, procedeu-se com a instalação do MUNGE:

```
sudo apt-get update
sudo apt-get install -y munge
```

4.2.2.2 Geração da Chave Criptográfica em Nó mestre

A arquitetura de segurança do MUNGE baseia-se em uma chave simétrica compartilhada. Esta chave deve ser gerada exclusivamente no nó mestre, da seguinte forma:

```
sudo create-munge-key -r
```

4.2.2.3 Distribuição e Aplicação de Permissões

A chave criptográfica gerada deve ser propagada do nó mestre para todos os nós trabalhadores por meio de um canal seguro. Para este caso, foi adotado o `scp`, que copia arquivos e pastas de forma segura entre computadores utilizando o protocolo *Secure Shell* (SSH). O exemplo abaixo demonstra a estrutura para se copiar um arquivo da máquina atual, nó mestre, para um destino generalizado.

```
scp /etc/munge/munge.key <usuario>@<ip>:<local_na_maquina_destino>
```

O MUNGE é um serviço estritamente sensível a permissões. Desta forma, ele se recusará a inicializar caso os privilégios do arquivo da chave estejam demasiadamente abertos, isto é, muito permissivos. Portanto, após a cópia da chave para os nós, as seguintes restrições de permissão e propriedade devem ser aplicadas:

```
sudo chown munge:munge /etc/munge/munge.key
sudo chmod 400 /etc/munge/munge.key
```

4.2.2.4 Inicialização e Validação do Serviço

Com as chaves devidamente distribuídas e protegidas, o serviço *daemon* do MUNGE é habilitado para inicializar junto ao sistema operacional e, em seguida, é iniciado em todos os nós:

```
sudo systemctl enable --now munge
sudo systemctl status munge
```

Para atestar a correta configuração e garantir que a comunicação criptografada entre os nós esteja operante, testes de validação de serviço e credencial são executados:

```
# Teste de codificacao e decodificacao local
munge -n | unmunge

# Teste de autenticacao remota (do Master para um Worker)
munge -n | ssh <ip_ou_hostname_do_worker> unmunge
```

O sucesso homologa a camada de confiança do *cluster*, permitindo o avanço seguro para a instalação do Slurm.

4.2.3 Gerenciamento de Filas e Escalonamento

O nó controlador central é a entidade responsável por manter o estado global do *cluster*, gerenciar as filas de submissão, escalonar as tarefas e monitorar a integridade dos nós de processamento. Toda a inteligência de alocação de recursos reside no *daemon* `slurmctld`. Para que este componente assuma a orquestração do ecossistema eficientemente, faz-se necessário um processo estruturado de provisionamento. Desse modo, as subseções seguintes descrevem as etapas operacionais para o estabelecimento deste processo, compreendendo a instalação dos pacotes do controlador, Seção 4.2.3.1, a definição das políticas e da topologia no arquivo de configuração global, Seção 4.2.3.2, e concluindo com a inicialização e validação do serviço, Seção 4.2.3.3.

4.2.3.1 Instalação dos Pacotes do Controlador

Nesta etapa, é feita a instalação do serviço do controlador e dos binários de cliente, os quais permitem a interação administrativa local com o *cluster*:

```
sudo apt-get update
sudo apt-get install -y slurmctld slurm-client
```

4.2.3.2 Configuração Global (`slurm.conf`)

O comportamento do *cluster* é definido no arquivo `/etc/slurm/slurm.conf`. Para um ambiente com GPU é mandatório configurar o escalonador para compreender os recursos consumíveis (*Trackable Resources* (TRES) e *Generic Resources* (GRES)), permitindo a correta alocação de GPUs, CPUs e memória *Random Access Memory* (RAM) entre múltiplos *jobs*. Os parâmetros críticos inseridos no arquivo de configuração são:

```
# Parametros estruturais do slurm.conf
ClusterName=brasil6g_hpc
SlurmctldHost=<hostname_do_master>(<ip_do_master>)
MpiDefault=none
ReturnToService=1

# Configuracao de Escalonamento e Recursos
SelectType=select/cons_tres
SelectTypeParameters=CR_Core_Memory
GresTypes=gpu

# Definicao dos Nos e Particoes
NodeName=<hostname_worker> CPUs=112 Boards=1 SocketsPerBoard=2 \
```

```
CoresPerSocket=56 ThreadsPerCore=1 RealMemory=2000000 Gres=gpu:8
State=UNKNOWN
PartitionName=hpc_gpu Nodes=<hostname_worker> Default=YES MaxTime=
INFINITE State=UP
```

4.2.3.3 Inicialização do Controlador

Após a criação do arquivo de configuração e a confirmação de que o serviço MUNGE está operante, o controlador é habilitado e iniciado:

```
sudo systemctl enable --now slurmctld
sudo systemctl status slurmctld
```

Os dados a serem inseridos na definição de cada nó podem ser gerados com o comando `slurmd -C`.

4.2.4 Execução de Tarefas

Os nós atuam como o poder computacional bruto do *cluster*. Eles executam o *daemon* `slurmd`, que aguarda as instruções do controlador para iniciar os *jobs*, aplicar restrições de grupo e validar a saúde do servidor. Para que esses recursos de processamento sejam efetivamente integrados à malha de orquestração, exige-se uma configuração padronizada em cada servidor. Desse modo, as subseções a seguir detalham as etapas operacionais de provisionamento destes nós trabalhadores, iniciando pela instalação dos pacotes e sincronização das diretrizes do controlador, Seção 4.2.4.1, na sequência, aborda-se o mapeamento físico dos recursos de *hardware*, com ênfase na declaração das GPUs por meio do arquivo `gres.conf`, Seção 4.2.4.2, culminando na inicialização dos serviços e na validação operacional de todo o *cluster*, Seção 4.2.4.3.

4.2.4.1 Instalação e Sincronização de Configurações

Em nós trabalhadores, são necessárias as instalações apenas do *daemon* de execução e das bibliotecas cliente, conforme descrito abaixo:

```
sudo apt-get update
sudo apt-get install -y slurmd slurm-client
```

Para a topologia ser reconhecida, o arquivo `slurm.conf` gerado no nó mestre deve ser replicado de forma idêntica para o diretório `/etc/slurm/` de todos os nós trabalhadores. Esta cópia pode ser feita via SCP, seguindo a mesma abordagem já apresentada em 4.2.2.3.

4.2.4.2 Mapeamento Físico de GPU (`gres.conf`)

Para que o `slurmd` isole corretamente o acesso ao barramento *Peripheral Component Interconnect Express* (PCIe) e evite que dois *jobs* distintos colidam no uso de uma mesma GPU sem a devida permissão de alocação, foi necessário mapear manualmente os dispositivos lógicos correspondentes através do arquivo de configuração `/etc/slurm/gres.conf`, associando cada recurso genérico ao seu respectivo identificador de dispositivo:

```
# Mapeamento manual alternativo no gres.conf
NodeName=<hostname_worker> Name=gpu File=/dev/nvidia0
NodeName=<hostname_worker> Name=gpu File=/dev/nvidia1
```


4.2.4.3 Inicialização e Validação do *Cluster*

Com as configurações replicadas, o serviço de execução é ativado da seguinte forma:

```
sudo systemctl enable --now slurmd
sudo systemctl status slurmd
```

A validação final da arquitetura é realizada a partir do nó mestre, verificando se os servidores se reportaram corretamente e se os recursos gráficos foram alocados e reconhecidos pelo escalonador:

```
# 1. Verificacao do servico do Slurm
systemctl status slurmd

# 2. Verificacao de recursos alocados (GRES)
scontrol show node <hostname> | grep -i gres

# 3. Status da particao e dos nos
sinfo
```

4.2.5 Isolamento de Ambiente e *Runtimes* (Módulos e *Apptainer*)

Em ambientes de HPC multiusuário, é imperativo prover mecanismos que permitam a coexistência de múltiplas versões de *softwares*, compiladores e bibliotecas sem gerar conflitos de dependências no sistema operacional hospedeiro. Para solucionar este desafio na arquitetura, esta POC homologou a utilização combinada do sistema de módulos e isolamento via containerização com Apptainer (evolução do projeto Singularity) [15]. Para detalhar a operacionalização dessa estratégia, as subseções seguintes descrevem primeiramente os procedimentos de instalação do Lmod, na Seção 4.2.5.1, e do Apptainer, na Seção 4.2.5.2. Em seguida, a integração de ambas as ferramentas, demonstrando o seu uso combinado na submissão de tarefas no *cluster*, apresentada na Seção 4.2.5.3.

4.2.5.1 Instalação do Lmod

O Lmod ou *Load Modules*, desenvolvido na linguagem Lua, é usado para o gerenciamento dinâmico do ambiente de usuário, manipulando variáveis como `PATH` e `LD_LIBRARY_PATH` sob demanda. Sua instalação no Ubuntu 24.04 é realizada através dos repositórios oficiais da distribuição:

```
sudo apt-get update
sudo apt-get install -y lmod
```

Após a instalação, a infraestrutura exige estruturar uma hierarquia de diretórios onde os *scripts* de instrução de cada *software* serão armazenados.

4.2.5.2 Instalação do Apptainer (Singularity)

Para cargas de trabalho de IA, que frequentemente exigem um ecossistema complexo e engessado de dependências, o Apptainer atua como ferramenta de containerização. Diferentemente do Docker, o Apptainer é projetado nativamente para HPC. Desta forma, ele executa os contêineres no espaço do usuário, mitigando riscos de escalonamento de privilégios e herdando diretamente as restrições de *cgroups* impostas pelo Slurm.

A instalação no Ubuntu é conduzida adicionando o repositório oficial do projeto para garantir a versão mais recente com suporte à integração de GPUs:

```
sudo apt-get install -y software-properties-common
sudo add-apt-repository -y ppa:apptainer/ppa
sudo apt-get update
sudo apt-get install -y apptainer
```

O comando abaixo ilustra a forma de realizar o *download* e conversão de uma imagem de contêiner Docker para o formato .sif, usado pelo Apptainer:

```
apptainer pull cuda_test.sif docker://nvidia/cuda:12.2.2-base-ubuntu22.04
```

4.2.5.3 Uso Combinado na Submissão de Tarefas

A sinergia entre o gerenciador de recursos Slurm, a preparação do ambiente (Lmod) e o encapsulamento de imagens (Apptainer) é evidenciada no momento da submissão de um *job*. O *script* em lote (tradicionalmente submetido via comando *sbatch*) orquestra estas três camadas sequencialmente. O exemplo a seguir ilustra um *script* clássico de submissão para uma carga de trabalho de IA acelerada por GPU:

```
#!/bin/bash
#SBATCH --job-name=treinamento_ia
#SBATCH --partition=hpc_gpu
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --gres=gpu:1           # Solicita 1 GPU fisica para o job
#SBATCH --time=12:00:00
#SBATCH --output=logs/treinamento_%j.out

module purge
module load cuda/12.4
module load openmpi/4.1.6

apptainer exec --nv cuda_test.sif python3 treinar_modelo.py
```

Neste fluxo operacional, o Slurm isola as GPUs via *cgroups* e informa os IDs físicos permitidos. O comando *module load* prepara o ambiente subjacente do hospedeiro, enquanto o *apptainer exec* injeta seletivamente as bibliotecas e os *devices* */dev/nvidiaX* autorizados para o interior da imagem (*cuda_test*), garantindo assim desempenho nativo com total isolamento de dependências.

4.3 Cenário B: Ecossistema *Cloud-Native* e Fatiamento Virtual

Em contraponto à rigidez estrutural e ao acoplamento físico do ambiente *bare-metal* detalhado no Cenário A, o Cenário B explora a transição para o paradigma *cloud-native* [39]. Este ecossistema foi projetado para oferecer máxima flexibilidade operacional, resiliência e escalabilidade dinâmica por meio da virtualização e da orquestração avançada de contêineres.

O objetivo central desta prova de conceito é demonstrar como a sobreposição de camadas de abstração permite que múltiplos pesquisadores e diferentes cargas de trabalho de IA coexistam de forma segura e isolada [40]. Além da gestão de recursos brutos, este cenário valida a implantação de escalonadores modernos para processamento em lote (*batch-scheduling*) sobre o Kubernetes [41], essenciais para viabilizar fluxos científicos complexos e plataformas de alto nível voltadas para a produtividade.

Para construir e validar este ecossistema distribuído de forma coesa, a implantação metodológica segue um fluxo de provisionamento estruturado em camadas. A fundação inicia-se com a virtualização da infraestrutura através do hipervisor Proxmox, Seção 4.3.1, seguida imediatamente pelo mapeamento direto dos aceleradores físicos para as instâncias virtuais por meio do GPU *Passthrough*, Seção 4.3.2. Sobre essa base virtualizada, estabelece-se a malha de orquestração do Kubernetes, Seção 4.3.3, que ganha a capacidade de reconhecer e gerenciar recursos gráficos após a injeção do NVIDIA *Device Plugin*, Seção 4.3.4. Com o *cluster* operante e ciente do *hardware*, o escalonador Volcano, Seção 4.3.5, é introduzido para suprir as demandas clássicas de processamento massivo e fatiamento virtual de GPUs. Por fim, a arquitetura é complementada com as plataformas de MLOps voltadas ao usuário final, contemplando o provisionamento do JupyterHub para desenvolvimento interativo, Seção 4.3.6, e do KubeFlow para a orquestração de *pipelines* completos de ML, Seção 4.3.7.

4.3.1 Virtualização da Infraestrutura (Proxmox)

A fundação do ecossistema proposto baseia-se na abstração do *hardware* físico através do hipervisor *Proxmox Virtual Environment* (PVE) [42]. A adoção desta plataforma de virtualização, baseada em *Kernel-based Virtual Machine* (KVM), permite o particionamento seguro dos recursos, isolando de forma lógica os nós que comporão o *cluster* Kubernetes, como ilustra a Figura 11.

4.3.1.1 Imagem base, CPU, memória e GPU

Para a padronização do sistema operacional, adotou-se a imagem Ubuntu 24.04, na qual as informações de recursos alocados seguiram a recomendação mínima oficial do K8S, considerando a quantidade de memória maior ou igual à da GPU mapeada. Especificações registradas na Tabela 1.

Recurso	Nó Mestre	Nó Trabalhador
vCPU	4 vCPUs	8 vCPUs
Memória RAM	4 GB	16 GB
Disco (SO)	100 GB SSD	100 GB SSD
Sistema Operacional	Ubuntu 24.04 LTS	Ubuntu 24.04 LTS
<i>kernel</i>	6.8.0-31-generic	6.8.0-31-generic

Tabela 1: Especificações das VMs do *cluster* Kubernetes.

Com a fundação devidamente provisionada e acessível via rede, o passo subsequente consistiu na transposição das interfaces PCIe das GPUs físicas para o interior da instância.

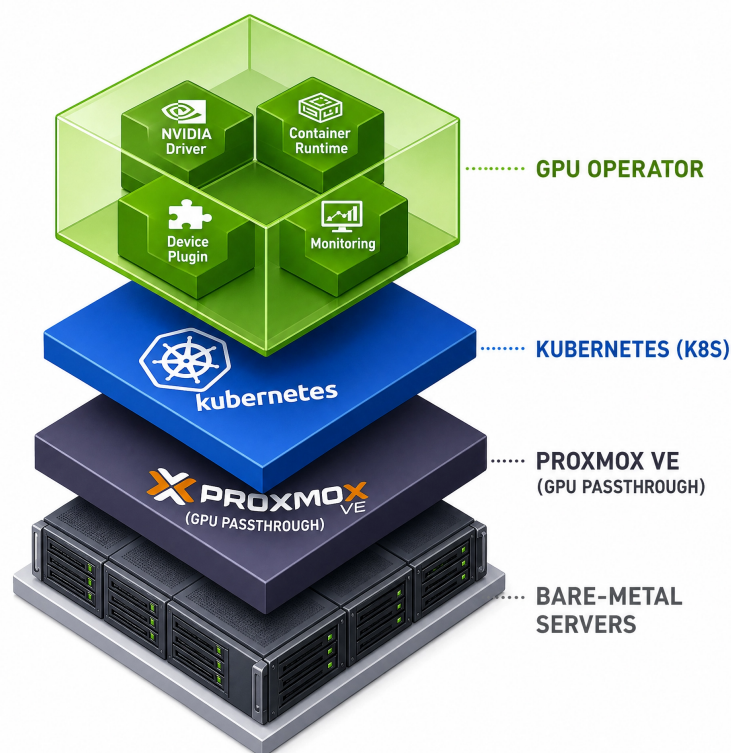


Figura 11: Ilustração das camadas de virtualização e containerização.

4.3.2 Mapeamento de GPU (*Passthrough*)

A configuração do GPU *Passthrough* é uma etapa técnica fundamental para garantir que o *cluster* Kubernetes tenha acesso direto ao *hardware* de aceleração gráfica. Após a fundação das instâncias estar devidamente provisionada e acessível via rede, o procedimento consiste na transposição das interfaces PCIe das GPUs físicas para o interior da instância virtualizada.

Este processo permite que o Proxmox VE ignore a camada de abstração do hipervisor para esse dispositivo específico, concedendo ao sistema operacional convidado (no caso, o Ubuntu 24.04) controle direto e quase nativo sobre o *hardware* da GPU. Essa configuração posiciona o Proxmox VE como a camada base de virtualização, estabelecendo a ponte necessária para que, acima dele, o Kubernetes possa utilizar o NVIDIA GPU *Operator* para orquestrar os recursos de maneira eficiente. A configuração segue os seguintes passos:

1. Configuração na *Basic Input/Output System* (BIOS): ativação de tecnologias como Intel *Virtualization Technology for Directed I/O* (VT-d) ou AMD *Virtualization for Directed I/O* (AMD-Vi);
2. Ativação do *Input/Output Memory Management Unit* (IOMMU): adição dos parâmetros de *kernel* no *GRand Unified Bootloader* (GRUB);
3. Módulos *Virtual Function I/O* (VFIO): configuração do carregamento dos módulos `vfio` para isolamento do *hardware*;
4. Identificação da GPU: mapeamento do ID do dispositivo via `lspci -nn` e vinculação ao *driver* `vfio-pci`;
5. Ajuste da máquina virtual: inclusão do dispositivo PCIe na configuração de *hardware* da máquina virtual.

Este processo permite que o sistema operacional convidado interaja diretamente com o *hardware*, viabilizando que o NVIDIA GPU Operator orquestre os recursos necessários dentro do Kubernetes.

4.3.3 Orquestração de contêineres (Kubernetes)

Com a infraestrutura virtualizada pronta e as GPUs devidamente mapeadas para o interior da instância, a próxima etapa consistiu na implantação do orquestrador Kubernetes. O procedimento de instalação seguiu as diretrizes padronizadas para implantações via `kubeadm` [43], com um ajuste arquitetural na escolha do *plugin* de rede (*Container Network Interface* (CNI)), no qual optou-se pela solução Flannel devido à sua simplicidade. Essa escolha também reflete a maturidade e a ampla adoção do Flannel em ambientes de produção, características que reduzem o risco de incompatibilidades durante a integração com os demais componentes da pilha.

Para consolidar essa arquitetura, a implantação foi estruturada em um fluxo operacional progressivo, detalhado nas subseções a seguir. O processo inicia-se com a preparação do sistema operacional, Seção 4.3.3.1, e a instalação do contêiner *runtime* com seus respectivos componentes, Seção 4.3.3.2. Posteriormente, descreve-se a inicialização do *Control Plane*, Seção 4.3.3.3, e a implantação do *Container Network Interface*, Seção 4.3.3.4, culminando no ingresso dos nós trabalhadores, Seção 4.3.3.5, para formar o *cluster* funcional.

4.3.3.1 Preparação do Sistema Operacional

Para garantir o funcionamento do Kubernetes, três configurações em nível de sistema são mandatórias em todos os nós e estão listadas abaixo:

- Módulos de *kernel* e roteamento *Internet Protocol* (IP): foram carregados os módulos `overlay` e `br_netfilter`, além da ativação do encaminhamento de pacotes IPv4 (`net.ipv4.ip_forward = 1`) no arquivo `/etc/sysctl.d/k8s.conf`, permitindo que o tráfego de rede passe do *host* para os contêineres.
- Configuração do *Domain Name System* (DNS): todos os nós do *cluster* precisam se comunicar tanto por IP como por nome. Neste caso, para atender a esse requisito, foi necessário adicionar as entradas IP e o nome das máquinas no arquivo `/etc/hosts` de cada máquina membro do *cluster*;
- Desativação da memória virtual de paginação (*swap*): o Kubernetes exige o desligamento da memória virtual de paginação para garantir a previsibilidade no consumo de RAM e forçar o comportamento correto do *Out of Memory Killer*. Isto foi feito através do comando `swapoff -a` e da remoção permanente da entrada no `/etc/fstab`;

A aplicação destas configurações foi realizada como exemplificado a seguir:

```
# 1. Carregamento de modulos de Kernel
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF
sudo modprobe overlay
sudo modprobe br_netfilter

# 2. Ativacao do encaminhamento IPv4
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
```

```
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
EOF
sudo sysctl --system

# 3. Desativacao do Swap
sudo swapoff -a
# Nota: Remova a entrada de swap permanentemente em /etc/fstab
```

4.3.3.2 Instalação do Contêiner *Runtime* e Componentes

Como motor de execução de contêineres, adotou-se o `containerd`, instalado e configurado como mostrado a seguir:

```
# 1. Instalacao do motor
sudo apt-get update && sudo apt-get install -y containerd.io

# 2. Geracao da configuracao default
sudo mkdir -p /etc/containerd
sudo containerd config default | sudo tee /etc/containerd/config.toml

# 3. Ajuste critico: SystemdCgroup = true
# Edite o arquivo /etc/containerd/config.toml e certifique-se de que:
# [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.
#   options]
#   SystemdCgroup = true

# 4. Aplicacao das configuracoes
sudo systemctl restart containerd
sudo systemctl enable containerd
```

Após a instalação do motor e a edição do arquivo `config.toml`, com a alteração da linha `SystemdCgroup = true`, foram realizadas a adição dos repositórios oficiais do Kubernetes e a instalação dos componentes principais do orquestrador:

```
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

A diretiva `apt-mark hold` foi utilizada como medida de segurança operacional, impedindo que os pacotes do Kubernetes sejam atualizados acidentalmente, o que poderia corromper o *cluster*.

4.3.3.3 Inicialização do *Control Plane* (Nó Mestre apenas)

A inicialização do *cluster* foi executada via `kubeadm`. Devido à decisão arquitetural de utilizar o Flannel como CNI, foi necessário declarar a faixa de IP `10.244.0.0/16` para a rede interna dos *Pods*, garantindo a compatibilidade nativa com os manifestos do Flannel [44]. Essa faixa de endereços é reservada exclusivamente para a comunicação entre *Pods*, permanecendo isolada da rede física dos nós, o que evita conflitos de roteamento com a infraestrutura de rede já existente no *data center*, como exibido a seguir:


```
sudo kubeadm init --pod-network-cidr=10.244.0.0/16

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

4.3.3.4 Implantação do Container *Network Interface*

Diferindo da abordagem com Calico ou Weave Net, a malha de rede L3/L4 foi estabelecida implantando o Flannel [44]. Esta etapa assegura que os *Pods* implantados em *workers* diferentes possam se comunicar de forma transparente através da rede subjacente:

```
kubectyl apply -f \
https://github.com/flannel-io/flannel/releases/latest/download/kube-
flannel.yml
```

4.3.3.5 Ingresso dos Nós trabalhadores

Com o *Control Plane* e a rede operantes, os nós *workers*, incluindo aqueles provisionados com *passthrough* de GPU, foram unificados ao *cluster*, utilizando o *token* seguro e o *hash* criptográfico, do tipo *Secure Hash Algorithm 256-bit* (SHA256), do certificado de autoridade gerados durante o *kubeadm init*, da seguinte forma:

```
sudo kubeadm join <IP_DO_MASTER>:6443 --token <TOKEN> \
--discovery-token-ca-cert-hash sha256:<HASH>
```

O sucesso da operação foi validado no nó mestre através do comando *kubectyl get nodes*, mostrado na Figura 12. Desta forma, confirma-se a transição dos nós para o estado *Ready*.

```
glauber@k8s-master-01:~/manifestos$ kubectyl get nodes
NAME          STATUS    ROLES    AGE      VERSION
k8s-master-01 Ready    control-plane  9m48s    v1.30.14
k8s-worker-01 Ready    <none>      3m53s    v1.30.14
k8s-worker-02 Ready    <none>      3m47s    v1.30.14
glauber@k8s-master-01:~/manifestos$ kubectyl get pods --all-namespaces
NAMESPACE     NAME                                     READY   STATUS    RESTARTS   AGE
kube-flannel   kube-flannel-ds-6lczm                 1/1     Running   0           45s
kube-flannel   kube-flannel-ds-952rq                 1/1     Running   0           45s
kube-flannel   kube-flannel-ds-hdbb7                 1/1     Running   0           45s
kube-system    coredns-55cb58b774-ljrsh              1/1     Running   0           9m4
kube-system    coredns-55cb58b774-xbgbk              1/1     Running   0           9m44s
kube-system    etcd-k8s-master-01                    1/1     Running   0           10m
kube-system    kube-apiserver-k8s-master-01           1/1     Running   0           10m
kube-system    kube-controller-manager-k8s-master-01 1/1     Running   0           10m
kube-system    kube-proxy-8zj5d                       1/1     Running   0           4m2s
kube-system    kube-proxy-kfp9t                       1/1     Running   0           9m44s
kube-system    kube-proxy-mb5s9                       1/1     Running   0           4m8s
kube-system    kube-scheduler-k8s-master-01           1/1     Running   0           10m
```

Figura 12: Listando os componentes do *cluster* K8S criado.

Com a base orquestradora finalizada, o ecossistema encontra-se apto para receber a integração ao nível dos contêineres com as GPUs via NVIDIA *Device Plugin*.

4.3.4 Descoberta e Integração de GPU (NVIDIA *Device Plugin*)

Nativamente, o Kubernetes é projetado para rastrear e alocar recursos computacionais padronizados, como CPU e memória RAM. Para o orquestrador ser capaz de descobrir, monitorar e fatiar as GPUs, é imperativa a implantação de um *plugin* disponibilizado pela NVIDIA. A integração entre o *hardware* e o ecossistema *cloud-native* foi realizada através da implantação combinada do NVIDIA Container Toolkit no nível do *host* (Máquina Virtual), e do NVIDIA Device Plugin no nível do *cluster* (K8S) [45]. O fluxo de implantação foi estruturado em duas etapas críticas para garantir que o *runtime* de contêineres e o *kubelet* gerenciem corretamente as cargas de trabalho. Desse modo, as subseções seguintes detalham, inicialmente, a preparação do *host*, englobando a instalação do *driver* e do NVIDIA Container Toolkit, Seção 4.3.4.1, para, em seguida, abordar a implantação do NVIDIA Device Plugin por meio de um *DaemonSet*, Seção 4.3.4.2, assegurando a visibilidade das GPUs dentro do K8S.

4.3.4.1 Preparação do *Host: Driver e Container Toolkit*

No nó que hospeda a GPU, a primeira etapa consistiu na instalação do *driver* proprietário da NVIDIA, seguindo o mesmo rigor metodológico detalhado no Cenário A. Com o *driver* instalado, foi realizada a instalação do NVIDIA *Container Toolkit* [46]. Passos descritos abaixo:

```
curl -fsSL \
https://nvidia.github.io/libnvidia-container/gpgkey | \
sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-
keyring.gpg

sudo apt-get update
sudo apt-get install -y nvidia-container-toolkit

sudo nvidia-ctk runtime configure --runtime=containerd

sudo systemctl restart containerd
```

Esta suíte de ferramentas atua como uma ponte, permitindo que os contêineres acessem o barramento da GPU. Como o ambiente utiliza o *containerd* como motor de execução, foi necessário reconfigurá-lo para reconhecer o *runtime* da NVIDIA e, em seguida, reiniciar o serviço.

4.3.4.2 Implantação do NVIDIA *Device Plugin* via *DaemonSet*

Com o *host* capacitado para executar contêineres com acesso à GPU, o passo seguinte foi instruir o plano de controle do Kubernetes a inventariar os recursos e disponibilizá-los para agendamento. Tarefa realizada aplicando o manifesto oficial do NVIDIA *Device Plugin* como *DaemonSet*, garantindo que o *plugin* rode automaticamente em todos os nós equipados com GPUs, inclusive naqueles adicionados posteriormente ao *cluster*. A implantação foi executada a partir do repositório oficial da NVIDIA [45]:

```
kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-device-
-plugin/v0.14.0/nvidia-device-plugin.yml
```

Para atestar o sucesso desta etapa, o comando `kubectl describe node k8s-worker-02` foi executado, confirmando a presença do novo recurso alocável conforme ilustrado na Figura 13, identificado como `nvidia.com/gpu`. Essa disponibilização é o requisito fundamental para que o escalonador reconheça e contabilize corretamente a capacidade de GPU de cada nó ao decidir o posicionamento de novas cargas de trabalho.

```
glauber@k8s-master-01:~$ kubectl get pods --all-namespaces
NAMESPACE      NAME                                     READY   STATUS    RESTARTS   AGE
kube-flannel    kube-flannel-ds-6lczm                  1/1     Running   0           4h57m
kube-flannel    kube-flannel-ds-952rq                  1/1     Running   0           4h57m
kube-flannel    kube-flannel-ds-hdbb7                  1/1     Running   1 (9m57s ago) 4h57m
kube-system     coredns-55cb58b774-ljrsh              1/1     Running   0           5h6m
kube-system     coredns-55cb58b774-xgbgk              1/1     Running   0           5h6m
kube-system     etcd-k8s-master-01                    1/1     Running   0           5h6m
kube-system     kube-apiserver-k8s-master-01           1/1     Running   0           5h6m
kube-system     kube-controller-manager-k8s-master-01 1/1     Running   0           5h6m
kube-system     kube-proxy-8zj5d                      1/1     Running   1 (9m57s ago) 5h
kube-system     kube-proxy-kfp9t                      1/1     Running   0           5h6m
kube-system     kube-proxy-mb5s9                      1/1     Running   0           5h
kube-system     kube-scheduler-k8s-master-01           1/1     Running   0           5h6m
kube-system     nvidia-device-plugin-daemonset-b5kd6   1/1     Running   0           16s
kube-system     nvidia-device-plugin-daemonset-fzgb4   1/1     Running   0           16s

glauber@k8s-master-01:~$ kubectl apply -f gpu-pod-test.yml
pod/gpu-pod created

glauber@k8s-master-01:~$ kubectl get pods --all-namespaces -o wide
NAMESPACE      NAME      READY   STATUS    RESTARTS   AGE
default        gpu-pod   0/1     Completed   0           3s

glauber@k8s-master-01:~$ kubectl logs gpu-pod
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
Done
```

Figura 13: Deploy do NVIDIA device plugin DaemonSet.

A partir desta configuração, qualquer manifesto submetido ao *cluster* contendo a diretiva `runtimeClassName:nvidia` e solicitando o recurso `nvidia.com/gpu` será corretamente agendado pelo *Kube-scheduler* para o nó compatível, completando assim a integração entre a camada de *runtime*, o plano de controle e o processo de alocação de recursos.

4.3.5 Gerenciamento de Filas e Fatiamento Virtual (Volcano)

O agendador padrão do Kubernetes é arquitetado para a orquestração contínua de microsserviços, cujo modelo assume cargas de trabalho independentes, de curta duração e sem dependência de execução simultânea entre si. No entanto, para cargas de trabalho de HPC e treinamentos de modelos massivos de IA, o ecossistema exige capacidades avançadas que inexistem na versão padrão, como o *gang-scheduling* (garantia de que todos os *pods* de um *job* paralelo iniciem simultaneamente) e o gerenciamento justo de filas (*fair-share scheduling*). Sem esses mecanismos, *jobs* distribuídos correm o risco de disputas descoordenadas por recursos entre múltiplos times que compartilham o mesmo *cluster*, além de atrasos na conclusão dos treinamentos decorrentes da execução parcial e incompleta de processos paralelos interdependentes.

Um cenário comum sem *gang-scheduling*, por exemplo, é o de um *job* paralelo que consegue alocar apenas parte de seus *Pods* *workers*, permanecendo indefinidamente ocioso à espera dos recursos restantes, enquanto ainda consome memória e ciclos de CPU do *cluster*. Para suprir esta deficiência estrutural, a arquitetura do Cenário B incorporou o escalonador Volcano, um sistema maduro de processamento em lote incubado pela CNCF [21]. Além do escalonamento propriamente dito, o Volcano também disponibiliza mecanismos nativos de fatiamento virtual de GPUs, permitindo que múltiplas cargas de trabalho compartilhem um mesmo dispositivo físico de forma isolada e controlada. A fim de documentar a integração deste componente, as subseções subsequentes descrevem o seu fluxo de provisionamento. O processo inicia-se pela confirmação de requisitos e preparação do ambiente, Seção 4.3.5.1, seguida pela instalação do escalonador central Volcano, Seção 4.3.5.2, por fim, a implantação do Volcano Device Plugin, Seção 4.3.5.3, essencial para o mapeamento e o fatiamento virtual adequado dos recursos no *cluster*.

4.3.5.1 Confirmação de Requisitos e Preparação

A transição para o novo paradigma de escalonamento exige a validação prévia de requisitos operacionais críticos do *cluster*, como por exemplo:

- Confirmação de operação plena do *Control Plane* do Kubernetes, garantindo compatibilidade com APIs `batch/v1`;
- Validação de que os nós equipados com GPU já possuam o NVIDIA *Container Toolkit* configurado corretamente sobre o *runtime containerd*;

4.3.5.2 Instalação do Escalonador Central Volcano

A implantação do núcleo do escalonador, que abrange os contêineres *Admission Controller*, *ControllerManager* e o próprio *Scheduler*, foi conduzida de forma declarativa e integral, evitando a necessidade de configuração manual de cada componente individualmente. Adotou-se o manifesto *Yet Another Markup Language* (YAML) oficial, garantindo a criação de todos os CRDs nativos do projeto, bem como do *namespace* dedicado `volcano-system`, no qual todos os recursos do escalonador são isolados e gerenciados. Essa abordagem elimina o risco de inconsistências entre versões dos componentes, já que todos os objetos são criados a partir de uma única fonte de verdade mantida oficialmente pelo projeto. Os comandos a seguir realizam essa implantação: o primeiro aplica o manifesto ao *cluster*, criando todos os recursos do escalonador dentro do *namespace* `volcano-system`, enquanto o segundo lista esses recursos, permitindo verificar se todos os contêineres e objetos foram corretamente instanciados e encontram-se em execução:

```
kubectl create -f https://raw.githubusercontent.com/volcano-sh/
  volcano/v1.9.0/installer/volcano-development.yaml
kubectl get all -n volcano-system
```

4.3.5.3 Implantação do Volcano Device Plugin

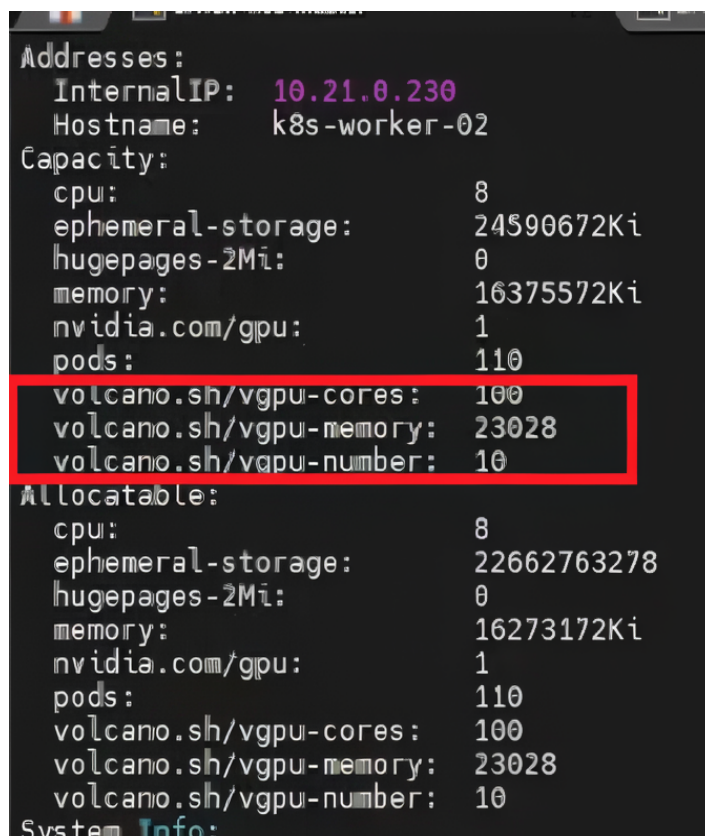
Para o orquestrador ser capaz de alocar GPUs de forma granular, aplica-se o manifesto mantido pela comunidade do projeto [47]. O Volcano *Device Plugin* opera como um *DaemonSet*, sendo instalado e verificado por meio dos comandos a seguir. O primeiro aplica o manifesto ao *cluster*, enquanto o segundo confirma a implantação bem-sucedida dos *Pods* no *namespace* `kube-system`:

```
kubectl apply -f https://raw.githubusercontent.com/volcano-sh/devices
/master/volcano-device-plugin.yml
kubectl get pods -n kube-system -l name=volcano-device-plugin
```

Com esta camada orquestradora finalizada e os *plugins* injetados, a infraestrutura *cloud-native* adquire o comportamento de um *cluster* HPC tradicional. A partir deste ponto, os *jobs* submetidos ao Kubernetes passam a exigir explicitamente a chave `schedulerName: volcano`, assegurando que a distribuição de processamento massivo seja administrada pelas regras de enfileiramento e localidade de dados do Volcano.

Além do escalonador de *jobs*, podemos mencionar a arquitetura obtida com a adoção do Volcano *Device Plugin*, o qual superamos a limitação nativa que tradicionalmente trata as GPUs como recursos indivisíveis (por exemplo, onde a requisição de `nvidia.com/gpu:1` monopoliza a placa inteira). Para ambientes de pesquisa e prototipagem, nos quais múltiplas inferências leves ou *notebooks* interativos concorrem pelo *hardware*, o fatiamento de GPU torna-se essencial, evitando a subutilização de aceleradores caros diante de cargas de trabalho que, individualmente, demandam apenas uma fração de sua capacidade total.

Através do Volcano, o orquestrador passa a expor os recursos de GPU virtuais, permitindo que os pesquisadores solicitem fatias baseadas na *Video Random Access Memory* (VRAM) disponível, como mostrado na Figura 14, maximizando a densidade de *pods* por nó. Essa granularidade também simplifica o processo de cobrança e contabilização de uso (*chargeback*), já que os recursos consumidos podem ser mensurados de forma proporcional à fração de GPU efetivamente alocada a cada projeto ou usuário.



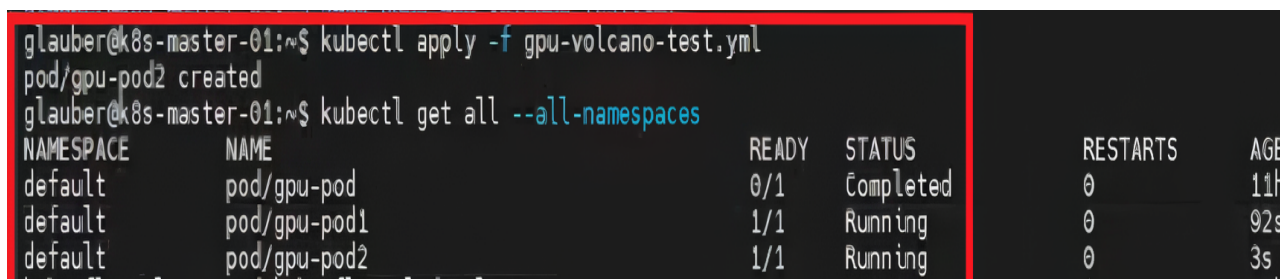
```
Addresses:
  InternalIP: 10.21.0.230
  Hostname: k8s-worker-02
Capacity:
  cpu: 8
  ephemeral-storage: 24590672Ki
  hugepages-2Mi: 0
  memory: 16375572Ki
  nvidia.com/gpu: 1
  pods: 110
  volcano.sh/vgpu-cores: 100
  volcano.sh/vgpu-memory: 23028
  volcano.sh/vgpu-number: 10
Allocatable:
  cpu: 8
  ephemeral-storage: 22662763278
  hugepages-2Mi: 0
  memory: 16273172Ki
  nvidia.com/gpu: 1
  pods: 110
  volcano.sh/vgpu-cores: 100
  volcano.sh/vgpu-memory: 23028
  volcano.sh/vgpu-number: 10
System Info:
```

Figura 14: Recursos de GPU disponibilizados de forma granular.

O fragmento de código YAML a seguir ilustra um manifesto de *pod* submetido ao *cluster*, no qual o usuário delega o agendamento ao Volcano e solicita apenas uma fração da GPU.

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-slice-workload
spec:
  schedulerName: volcano
  containers:
  - name: ai-model-training
    image: nvidia/cuda:12.4.1-base-ubuntu22.04
    command: ["nvidia-smi"]
    resources:
      limits:
        volcano.sh/vgpu-memory: 3000
        volcano.sh/vgpu-number: 1
```

Ao interceptar essa requisição, o *Scheduler* do Volcano avalia o inventário global e aloca o contêiner em um nó que possua a capacidade fracionada correspondente disponível. O *Device Plugin* então atua no nível do *host* para impor o isolamento da memória virtualizada, garantindo que o limite de 3000 MB não seja excedido pela aplicação, possibilitando que múltiplos pods usufruam do mesmo recurso conforme apresentado nas Figuras 15 e 16, consolidando assim a multi-alocação segura do ecossistema *cloud-native*.



```
glauber@k8s-master-01:~$ kubectl apply -f gpu-volcano-test.yml
pod/gpu-pod2 created
glauber@k8s-master-01:~$ kubectl get all --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	pod/gpu-pod	0/1	Completed	0	11h
default	pod/gpu-pod1	1/1	Running	0	92s
default	pod/gpu-pod2	1/1	Running	0	3s

Figura 15: Múltiplos *pods* em execução em uma mesma GPU.

4.3.6 Ambiente de Desenvolvimento Interativo (JupyterHub)

Com a fundação orquestrada e o fatiamento de GPUs operacionais, a arquitetura do Cenário B exige uma interface de alto nível que abstraia a complexidade do Kubernetes para os pesquisadores e cientistas de dados. Para este fim, o JupyterHub foi selecionado como o portal de acesso interativo padrão. Ele atua como um gerenciador de sessões multiusuário, provisionando dinamicamente contêineres individuais (*notebooks*) sob demanda para cada usuário autenticado, sem que este precise conhecer os detalhes de agendamento, alocação de recursos ou orquestração subjacentes ao *cluster*.

A implantação do JupyterHub no *cluster* foi realizada pelo Helm [48], que se trata de um gerenciador de pacotes para o K8S *cloud-native*. Esta abordagem garante um ciclo de vida controlado, facilitando atualizações, reversões (*rollbacks*) e a padronização das configurações por meio de arquivos de valores, o *values.yaml*.


```
glauber@k8s-master-01:~$ kubectl exec -it gpu-pod1 -- /bin/bash
root@gpu-pod1:/# nvidia-smi
[HAMi-core Msg(69:137066472347456:libvgpu.c:870)]: Initializing.....
Tue Jun 2 12:11:39 2026
```

NVIDIA-SMI 595.71.05			Driver Version: 595.71.05			CUDA Version: 13.2		
GPU	Name	Perf	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC
Fan	Temp		Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.	MIG M.
0	NVIDIA RTX A5000	P8	Off	00000000:00:10:0	Off	0%	Default	0
30%	24C		4W / 230W		0MiB / 3000MiB			N/A

```

Processes:
GPU  GI  CI          PID  Type  Process name                      GPU Memory
ID   ID  ID                               Usage
-----
No running processes found

[HAMi-core Msg(69:137066472347456:multiprocess_memory_limit.c:703)]: Cleanup on exit for PID 69
[HAMi-core Msg(69:137066472347456:multiprocess_memory_limit.c:739)]: Exit cleanup complete for PID 69
root@gpu-pod1:/# exit
exit
glauber@k8s-master-01:~$ kubectl exec -it gpu-pod2 -- /bin/bash
root@gpu-pod2:/# nvidia-smi
[HAMi-core Msg(59:127211750119232:libvgpu.c:870)]: Initializing.....
Tue Jun 2 12:11:59 2026
```

NVIDIA-SMI 595.71.05			Driver Version: 595.71.05			CUDA Version: 13.2		
GPU	Name	Perf	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC
Fan	Temp		Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.	MIG M.
0	NVIDIA RTX A5000	P8	Off	00000000:00:10:0	Off	0%	Default	0
30%	24C		4W / 230W		0MiB / 4000MiB			N/A

```

Processes:
GPU  GI  CI          PID  Type  Process name                      GPU Memory
ID   ID  ID                               Usage
-----
No running processes found

[HAMi-core Msg(59:127211750119232:multiprocess_memory_limit.c:703)]: Cleanup on exit for PID 59
[HAMi-core Msg(59:127211750119232:multiprocess_memory_limit.c:739)]: Exit cleanup complete for PID 59
root@gpu-pod2:/#
```

Figura 16: Pods em execução com fatias de tamanhos diferentes.

4.3.6.1 Preparação do Repositório e Configuração do *KubeSpawner*

O processo iniciou-se com a adição do repositório oficial do projeto Jupyter [49] ao cliente Helm. O núcleo da configuração reside no mapeamento entre o *KubeSpawner* (módulo responsável por criar os *pods* dos usuários) e as capacidades do *cluster*.

Foi elaborado um arquivo de configuração (*config.yaml*) personalizado para injetar as especificações de *hardware* diretamente nos perfis de usuário. O fragmento abaixo ilustra a configuração para habilitar o fatiamento de GPU via Volcano:

```
singleuser:
  image:
    name: jupyter/pytorch-notebook
    tag: cuda12-latest
  extraPodConfig:
    schedulerName: volcano
```

```
runtimeClassName: nvidia
profileList:
- display_name: "Ambiente CPU Base"
  description: "Ideal para prototipacao leve e analise de dados."
  default: true
- display_name: "Ambiente IA - GPU Fracionada (4GB VRAM)"
  description: "Aceleracao grafica para inferencia e testes."
  kubespawner_override:
    extraResource:
      limits:
        volcano.sh/vgpu-memory: 4000
        volcano.sh/vgpu-number: 1
```

Essa abordagem declarativa permite que a interface do JupyterHub apresente um menu de seleção no momento do *login*, onde o usuário pode escolher a quantidade exata de aceleração necessária, otimizando o consumo global do *cluster*.

4.3.6.2 Implantação e Exposição do Serviço

Com os valores definidos, a implantação do serviço de infraestrutura *chart* foi executada em um *namespace* isolado, garantindo a segregação lógica de redes e recursos, conforme é ilustrado na Figura 17.

```
glauber@k8s-master-01:~$ kubectl get all -n jupyterhub
```

NAME	READY	STATUS	RESTARTS	AGE
pod/continuous-image-puller-767vp	1/1	Running	3 (73d ago)	228d
pod/continuous-image-puller-k97ts	1/1	Running	1 (73d ago)	117d
pod/hub-599c5fbf75-2tlx8	1/1	Terminating	3 (73d ago)	228d
pod/hub-599c5fbf75-tpsd	0/1	Pending	0	50d
pod/proxy-5486c8fd45-g98bl	0/1	Pending	0	50d
pod/proxy-5486c8fd45-n8c4l	1/1	Terminating	3 (73d ago)	228d
pod/user-scheduler-6c48f59b8-48ksh	1/1	Terminating	5 (73d ago)	228d
pod/user-scheduler-6c48f59b8-5j77c	0/1	Pending	0	50d
pod/user-scheduler-6c48f59b8-c5bbq	0/1	Pending	0	50d
pod/user-scheduler-6c48f59b8-w9gg7	1/1	Terminating	3 (73d ago)	228d

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/hub	ClusterIP	10.100.204.192	<none>	8081/TCP	228d
service/proxy-api	ClusterIP	10.97.240.190	<none>	8001/TCP	228d
service/proxy-public	NodePort	10.110.254.100	<none>	80:31733/TCP	228d

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR	AGE
daemonset.apps/continuous-image-puller	0	0	0	0	0	<none>	228d

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/hub	0/1	1	0	228d
deployment.apps/proxy	0/1	1	0	228d
deployment.apps/user-scheduler	0/2	2	0	228d

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/hub-599c5fbf75	1	1	0	228d
replicaset.apps/hub-65f8549d7f	0	0	0	228d
replicaset.apps/proxy-5486c8fd45	1	1	0	228d
replicaset.apps/user-scheduler-6c48f59b8	2	2	0	228d

NAME	READY	AGE
statefulset.apps/user-placeholder	0/0	228d

```
glauber@k8s-master-01:~$
```

Figura 17: Componentes do JupyterHub no K8S.

O comando `helm repo add` adiciona o repositório oficial do projeto ao gerenciador de pacotes local, enquanto o `helm upgrade -install` instala ou atualiza o *release*, aplicando de forma idempotente as configurações definidas no arquivo `config.yaml`. A flag `-cleanup-on-fail` garante que, em caso de falha durante a implantação, os recursos parcialmente criados sejam removidos automaticamente, evitando resíduos inconsistentes no *cluster*, enquanto `-create-namespace`

dispensa a criação prévia e manual do *namespace* de destino. Essa combinação de *flags* reflete um princípio importante de operação em ambientes *cloud-native*: a implantação deve ser idempotente e segura o suficiente para ser executada repetidamente, sem exigir intervenção manual de limpeza em caso de falha parcial durante o processo.

```
helm repo add jupyterhub https://jupyterhub.github.io/helm-chart/
helm repo update
helm upgrade --cleanup-on-fail --install jhub jupyterhub/jupyterhub
--namespace jupyter-system --create-namespace --values config.yaml
```

Para validar a integridade da entrega, um teste prático foi conduzido acessando a interface *web* gerada pelo serviço do JupyterHub, como mostrado na Figura 18. A disponibilidade dessa interface confirma que os componentes internos do *chart* — *hub*, *proxy* e os *pods* de *notebook* — foram corretamente inicializados e que a rede interna do *cluster* está corretamente configurada para expor o serviço aos usuários finais. Esse resultado encerra o ciclo de validação da plataforma, confirmando que o ambiente encontra-se apto a receber os pesquisadores para o desenvolvimento interativo de modelos.

Este teste de ponta a ponta é particularmente relevante neste contexto, pois valida simultaneamente três camadas distintas da arquitetura: o correto roteamento de rede interna promovido pelo Flannel, o funcionamento do *proxy* responsável por rotear as requisições de cada usuário ao seu respectivo *pod* de *notebook* e a integridade do próprio mecanismo de autenticação configurado no *KubeSpawner*. Um eventual erro em qualquer uma dessas camadas se manifestaria imediatamente como uma falha de acesso à interface, o que torna esta etapa um indicador abrangente e eficiente da saúde geral da implantação, sem a necessidade de inspecionar cada componente isoladamente. Dessa forma, um único teste de acesso substitui a necessidade de validações isoladas e sequenciais de cada camada da pilha tecnológica.

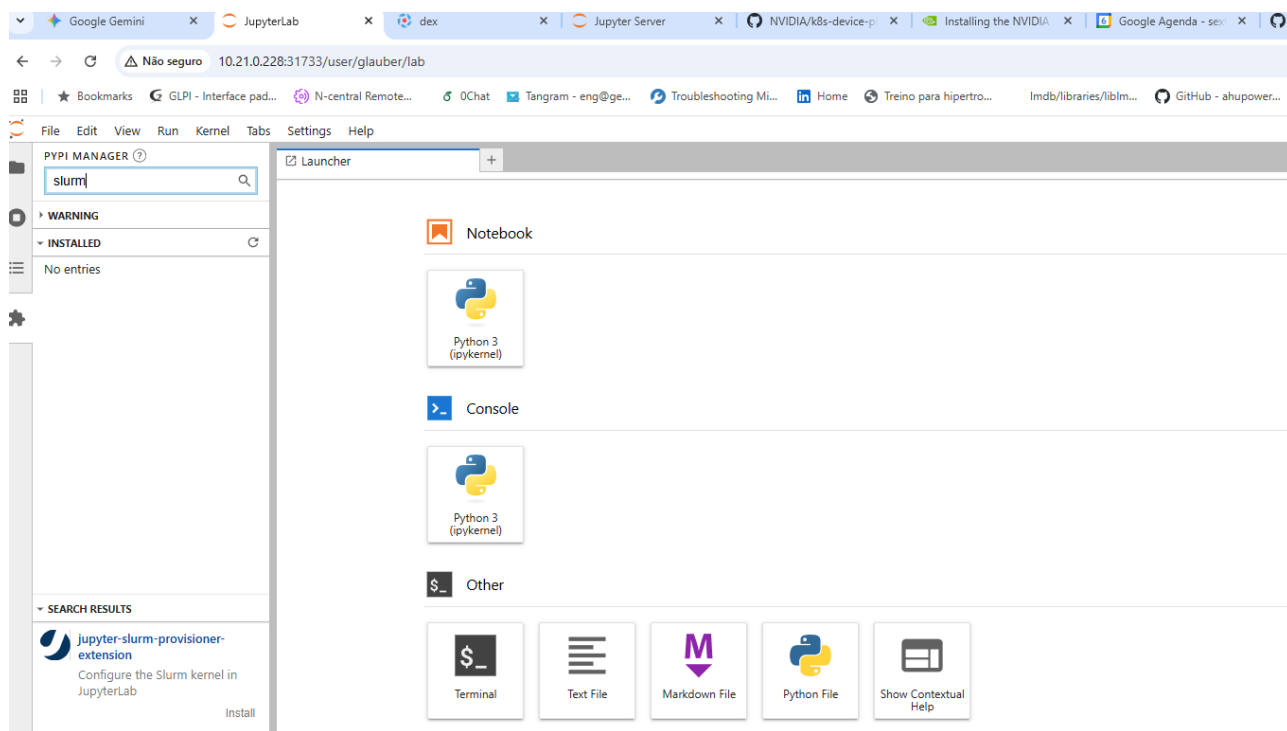


Figura 18: Interface *web* do JupyterHub.

4.3.7 Orquestração de *Pipelines* de ML (*Kubeflow*)

Enquanto o JupyterHub atende à necessidade de prototipagem interativa, modelos de IA exigem uma plataforma capaz de orquestrar fluxos de trabalho completos, desde o processamento de dados até o treinamento distribuído e a disponibilização do modelo. No Cenário B, este papel é desempenhado pelo Kubeflow [50], a principal plataforma *open-source* de MLOps nativa do Kubernetes. Sua adoção permite padronizar, sob uma única interface, etapas que de outra forma exigiriam ferramentas isoladas e integrações manuais entre si, reduzindo a fragmentação do fluxo de trabalho de ciência de dados.

A implantação do Kubeflow transforma o *cluster* em uma infraestrutura de ponta a ponta para ciência de dados. Devido à sua arquitetura baseada em múltiplos microsserviços fortemente acoplados (Istio, Knative, Katib e *pipelines*), a instalação não utiliza o Helm tradicional, mas sim o Kustomize, o qual é uma ferramenta de *deploy* do K8S que permite a sobreposição de configurações diretamente nos manifestos YAML oficiais [50]. Essa abordagem declarativa preserva a integridade das dependências entre os componentes, evitando conflitos de versão que poderiam surgir de instalações independentes de cada microsserviço.

4.3.7.1 Implantação via Kustomize

A implantação foi realizada a partir do repositório oficial de manifestos do Kubeflow. O processo exige a construção e aplicação hierárquica dos componentes, garantindo que as dependências de rede (como o *Service Mesh Istio*) estejam operantes antes dos serviços de aplicação. Como os CRDs de alguns componentes só se tornam disponíveis após a aplicação bem-sucedida de outros manifestos anteriores na mesma cadeia de dependências, o processo de implantação é executado de forma iterativa, conforme demonstrado a seguir: primeiro o repositório é clonado na versão v1.8.0, e em seguida um laço de repetição tenta aplicar continuamente a saída do `kustomize build`, aguardando 15 segundos entre cada tentativa, até que todos os recursos sejam reconhecidos e aceitos pelo *control plane* sem erros:

```
git clone -b v1.8.0 https://github.com/kubeflow/manifests.git
cd manifests
while ! kustomize build example | kubectl apply -f -; do \
    echo "Aguardando inicializacao dos CRDs..."; \
    sleep 15; \
done
```

4.3.7.2 Sinergia com o Escalonador Volcano e GPU

A principal justificativa técnica para a adoção do *Kubeflow* neste ecossistema é o seu *Training Operator*. Este operador introduz CRDs específicos para IA, como o *TFJob* para TensorFlow, e o *PyTorchJob*. A grande inovação arquitetural desta configuração é a integração nativa destes *Jobs* com o escalonador Volcano previamente instalado. Para treinamentos distribuídos que utilizam múltiplos nós simultaneamente para treinar um único modelo, o *Training Operator* delega o *gang-scheduling* ao Volcano, garantindo que o treinamento só inicie se houver recursos disponíveis para todos os nós paralelos.

O fragmento a seguir ilustra a submissão de um treinamento distribuído de PyTorch por meio do Kubeflow, requisitando o Volcano e as GPUs mapeadas no *cluster*. Destaca-se o campo `minAvailable: 2`, que instrui explicitamente o Volcano a aguardar a disponibilidade simultânea de ambas as réplicas antes de iniciar a execução do *Job*:

```

apiVersion: "kubeflow.org/v1"
kind: "PyTorchJob"
metadata:
  name: "treinamento-distribuido-brasil6g"
  namespace: "kubeflow-user"
spec:
  schedulingPolicy:
    schedulerName: volcano
    minAvailable: 2
  pytorchReplicaSpecs:
    Master:
      replicas: 1
      template:
        spec:
          containers:
            - name: pytorch
              image: kubeflow/pytorch-mnist:latest
    Worker:
      replicas: 2
      template:
        spec:
          containers:
            - name: pytorch
              image: kubeflow/pytorch-mnist:latest
          resources:
            limits:
              nvidia.com/gpu: 1

```

4.3.8 Slinky

Embora o Kubeflow e o Volcano atendam perfeitamente aos fluxos de trabalho nativos de contêineres, muitos pesquisadores, códigos legados e *scripts* científicos de simulação da camada física (PHY) ainda dependem da interface clássica do Slurm Workload Manager para submissão de tarefas. O projeto Slinky [22], mantido oficialmente pela SchedMD, soluciona essa lacuna ao viabilizar a execução integral do ecossistema Slurm dentro do Kubernetes.

A adoção do Slinky permite que o *cluster cloud-native* atue como um ambiente híbrido transparente: pesquisadores habituados ao paradigma *bare-metal* (Cenário A) podem continuar utilizando comandos como **sbatch** e **srun**, enquanto, nos bastidores, o processamento ocorre em contêineres e é gerenciado pelo Kubernetes.

1. Procedimento de Instalação

Seguindo as diretrizes oficiais de implantação da SchedMD, o provisionamento do Slinky é orquestrado de forma declarativa utilizando o gerenciador de pacotes Helm e o padrão *Operator* do Kubernetes. O processo de instalação divide-se na preparação de certificados, implantação do operador e, finalmente, a criação do *cluster* Slurm:

```

# Passo A: instalacao do cert-manager (pre-requisito para webhooks do
operador)

```



```
helm install cert-manager oci://quay.io/jetstack/charts/cert-manager
--namespace cert-manager --create-namespace --set crds.enabled=
true

# Passo B: Instalacao das CRDs (Custom Resource Definitions) do
Slinky
helm install slurm-operator-crds oci://ghcr.io/slinkyproject/charts/
slurm-operator-crds

# Passo C: Implantacao do Operador Slurm
helm install slurm-operator oci://ghcr.io/slinkyproject/charts/slurm-
operator --namespace=slinky --create-namespace

# Passo D: Implantacao do Cluster Slurm (Controlador e Workers)
helm install slurm oci://ghcr.io/slinkyproject/charts/slurm --
namespace=slurm --create-namespace
```

Esta abordagem modular, garante que as definições de recursos personalizados sejam corretamente compreendidas pela API do Kubernetes antes que o controlador do Slurm tente instanciá-los.

2. Arquitetura da Implantação e Gestão de Estado

A implantação do Slinky transcreve a topologia clássica do Slurm (Controlador e *workers*) para recursos nativos do Kubernetes (como *StatefulSets* e *DaemonSets*), como ilustra a Figura 19. Devido à natureza efêmera dos contêineres, o maior desafio na estabilização desta arquitetura consiste no mapeamento rigoroso do armazenamento.

O *slurmctld* (controlador) exige persistência absoluta de estado para não perder o rastreamento das filas e da bilhetagem em caso de reinicialização do *Pod*. Para garantir essa resiliência, a implantação exigiu o provisionamento de PVC acoplados a *StorageClasses* de alta disponibilidade no *cluster*, garantindo que diretórios críticos como */var/spool/slurm* e */etc/slurm* sobrevivessem ao ciclo de vida dos contêineres.

A operacionalização final revelou um fluxo de trabalho extremamente poderoso: a alocação dinâmica de nós *workers* do Slurm como *pods* efêmeros, que escalam automaticamente em resposta à fila de *jobs* submetidos. Uma vez processada a carga de trabalho, os *pods* correspondentes aos *workers* ociosos são destruídos.

4.4 Cenário C: Arquitetura de Referência

Os cenários anteriores detalharam arquiteturas elaboradas por meio de ferramentas baseadas em código aberto, como o ecossistema *cloud-native*. O Cenário C, em contrapartida, valida a implantação da arquitetura de referência oficial e homologada pela NVIDIA. Esta abordagem foi projetada para mitigar a complexidade operacional, automatizar o ciclo de vida da infraestrutura e oferecer suporte de nível corporativo.

Para demonstrar a agilidade de provisionamento e a fluidez operacional desta solução, as atividades apresentadas a seguir seguem um fluxo lógico de implantação centralizada. O núcleo desta arquitetura baseia-se no sistema BCM, Seção 4.4.1, iniciando-se pelo provisionamento de seu servidor de gerência, o *Head Node*, Seção 4.4.2. A partir dessa fundação, a capacidade computacional do *cluster* é escalada por meio da descoberta e adição automatizada de nós de

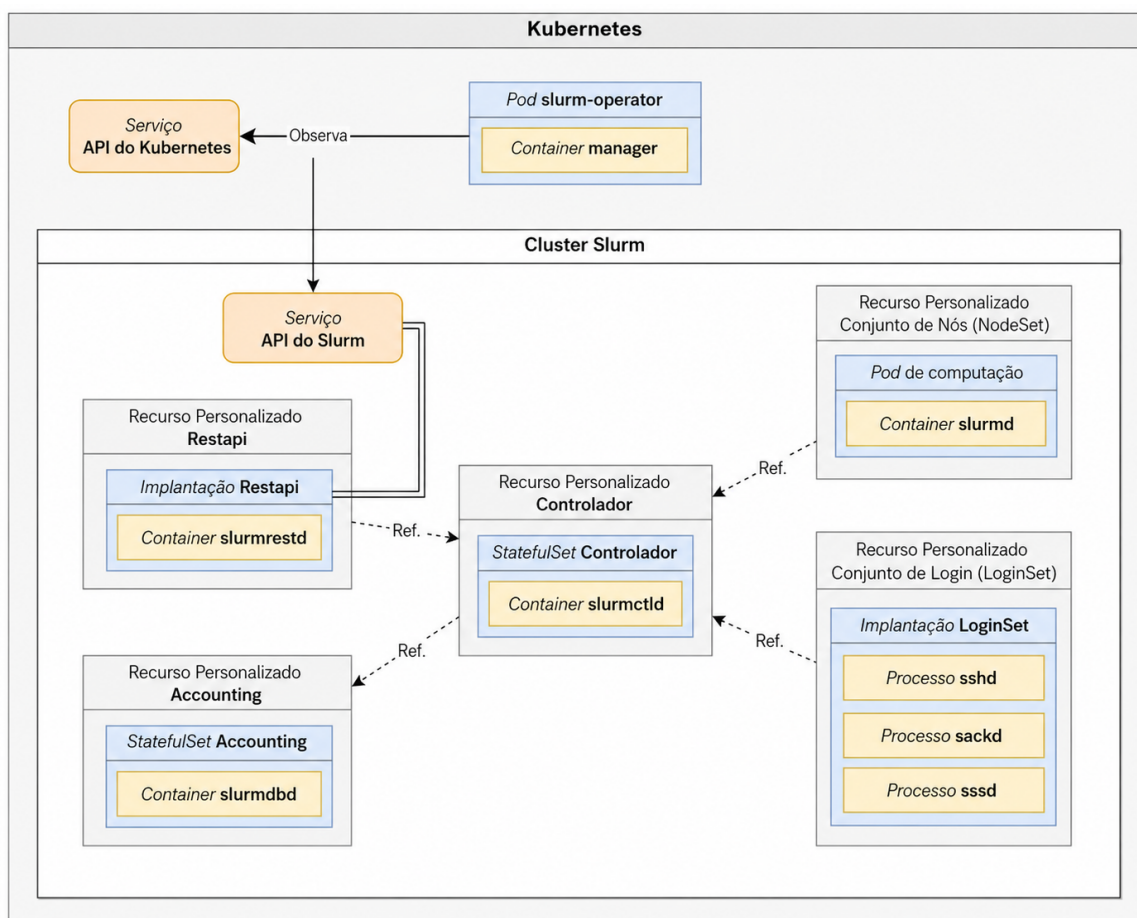


Figura 19: Arquitetura do Slinky (Gerado por IA).

processamento, Seção 4.4.3, e estendida para dispositivos de borda por meio do paradigma de *edge computing*, Seção 4.4.4. No nível de otimização de *hardware*, o cenário explora o particionamento direto das GPUs utilizando a tecnologia MIG, Seção 4.4.5. Por fim, a convergência tecnológica é consolidada por meio da implantação automatizada guiada pelos assistentes nativos do próprio BCM, os quais estabelecem a malha do orquestrador Kubernetes, Seção 4.4.6, e o portal de desenvolvimento interativo JupyterHub, Seção 4.4.7, entregando um ecossistema pronto para produção.

4.4.1 Sistema Central de Gerenciamento (BCM)

O *cluster* é gerenciado pelo BCM o qual introduz recursos nativos de convergência entre os ambientes *bare-metal* clássicos e as cargas de trabalho *cloud-native*. Esta ferramenta atua como o sistema nervoso central do *cluster*, automatizando o provisionamento, a configuração de redes, a distribuição de *drivers* otimizados e a monitoramento de *hardware*.

4.4.2 Instalação e Provisionamento do Nó Controlador (*Head Node*)

A implantação da *stack* corporativa inicia-se pelo servidor de controle primário. Diferentemente das implantações convencionais, no qual o sistema operacional e os serviços de gerência são instalados separadamente, o BCM centraliza esse fluxo através de uma mídia de instalação unificada, cujas etapas são detalhadas abaixo.

4.4.2.1 Obtenção da Mídia e Requisitos Iniciais

O processo de instalação demanda a aquisição da imagem oficial do instalador do BCM. Por tratar-se de um *software* da suíte NVAIE, o *download* restringe-se ao portal de licenciamento corporativo da NVIDIA. A imagem consolidada carrega consigo não apenas o assistente de instalação principal, mas também o conjunto de ferramentas CUDA e imagens de sistemas operacionais personalizados e homologados para distribuição subsequente aos nós de processamento [51].

4.4.2.2 Instalação e Configuração via *cm-installer*

Com a imagem montada, o processo de inicialização invoca diretamente o assistente de instalação. O fluxo de configuração abrange as seguintes etapas:

- **Particionamento Estratégico:** configuração das unidades de armazenamento do nó mestre, alocando *Logical Volume Manager* (LVM) para diretórios sensíveis como */cm* (dados e imagens do *cluster*) e o banco de dados interno de telemetria;
- **Definição da Topologia de Rede:** separação lógica das interfaces físicas, definindo explicitamente a rede de gerência externa para acesso administrativo e a rede interna de provisionamento. A rede interna é vital, pois o BCM assumirá os serviços de *Dynamic Host Configuration Protocol* (DHCP), DNS, e *Trivial File Transfer Protocol* (TFTP) para *boot* pela rede, visando gerenciar os nós trabalhadores automaticamente;
- **Seleção da Carga de Trabalho:** durante a instalação, o assistente permite pré-selecionar os ecossistemas de orquestração que o *cluster* utilizará. Para manter a paridade comparativa com os cenários anteriores, ativam-se as ramificações de gerenciadores de carga de trabalho, também conhecidos como *workload managers*, para suporte nativo ao Kubernetes e ao Slurm integrados.

4.4.2.3 Acesso Inicial e Administração via *Base View*

Concluído o processo de instalação, o servidor assume controle total sobre a infraestrutura designada. A homologação da instalação é feita através do acesso à interface administrativa gráfica denominada *Base View*, conforme ilustrado na Figura 20. O acesso inicial é realizado via navegador *web* (apontando para o IP do *Head Node*:8081/*base-view*), utilizando as credenciais de superusuário (*root*) configuradas durante a instalação.

A partir do painel de controle interativo, a equipe de infraestrutura obtém uma visão imediata do estado do servidor e dos demais serviços. É também por essa interface que se inicia o processo de provisionamento automático dos nós ao *cluster*.

4.4.3 Descoberta e Adição Automatizada de Nós de Processamento

Uma vez que o *Head Node* assume o controle da infraestrutura de rede interna de gerência, a adição de novos servidores de processamento ao *cluster* dispensa a instalação manual de sistemas operacionais. O BCM utiliza um mecanismo autônomo de descoberta de *hardware* associado à distribuição centralizada de imagens de *software* [51]. Essa abordagem elimina a necessidade de mídias de instalação físicas ou intervenção presencial junto ao *rack*, permitindo que a equipe técnica escale a capacidade computacional do *cluster* remotamente, a partir de uma única interface centralizada. Os passos a seguir descrevem os processos para a adição de nós.

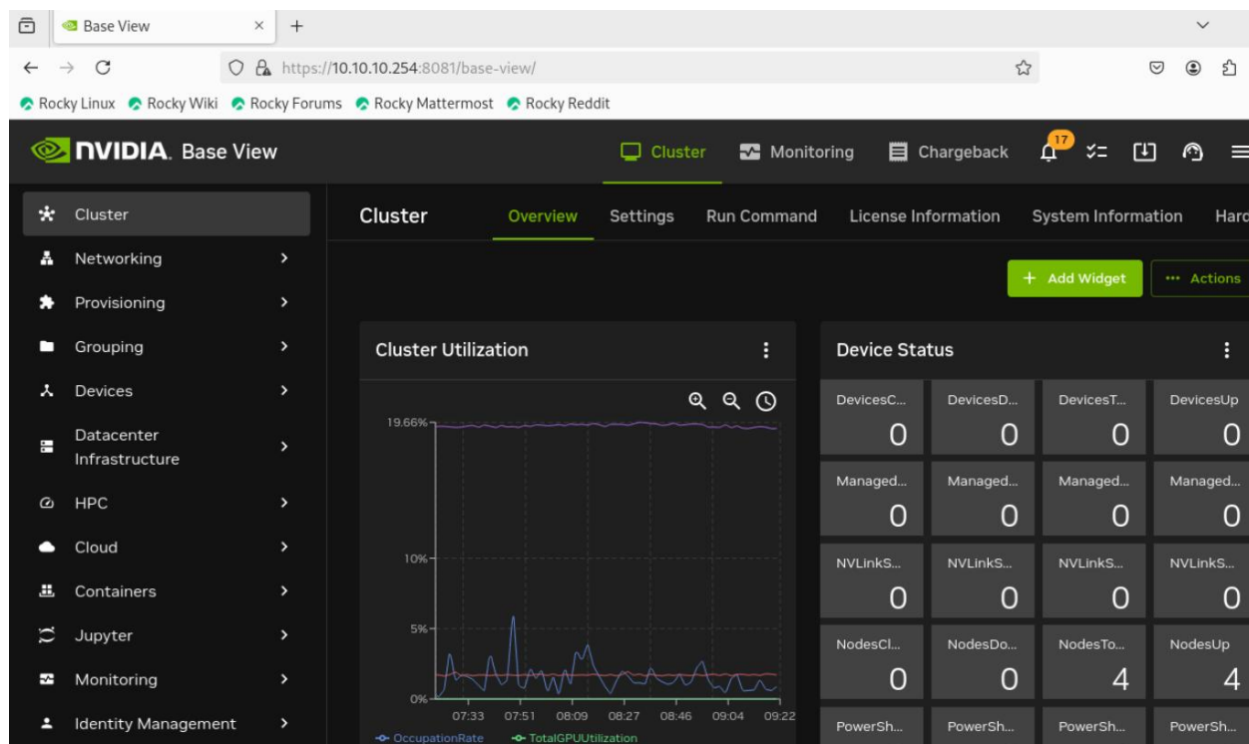


Figura 20: Interface do *Base View*.

4.4.3.1 Descoberta de nós (*Node Discovery* via PXE)

A fase de ingestão de novos nós baseia-se no protocolo de inicialização por rede *Preboot Execution Environment* (PXE). Sem possuir sistema operacional nos discos locais, a interface de rede do nó envia uma solicitação DHCP em *broadcast*. O *Head Node*, atuando como servidor DHCP autoritativo da malha interna, intercepta a requisição, captura o endereço *Media Access Control* (MAC) da interface física, e bloqueia temporariamente a inicialização do nó, alertando o administrador através da aba *Node Discovery* na interface gráfica de gestão (*Base View*). Esse bloqueio temporário é uma medida de segurança deliberada, evitando que um servidor não identificado seja incorporado automaticamente ao *cluster* sem a validação explícita de um administrador.

4.4.3.2 Categorização e Atribuição de Imagem de *software*

Com o endereço MAC capturado, o administrador cadastra o equipamento no inventário do *cluster*, atribuindo-lhe um *hostname* e um endereço IP estático interno. O BCM atribui o servidor a uma categoria de nó, o que permite aplicar mudanças de configuração a todos os nós daquela categoria de uma só vez, sem acesso individual a cada máquina.

4.4.3.3 Provisionamento Automático e Validação de Estado

Após a confirmação da categoria, o nó é autorizado a prosseguir com o *boot*. O servidor baixa um sistema operacional mínimo alocado em memória RAM, particiona os discos físicos conforme as diretrizes da categoria e realiza a sincronização integral da imagem de *software* a partir do nó principal.

Finalizada a cópia, o nó é reiniciado automaticamente, inicializando com a *stack* de *software* pronta para uso. Para atestar o sucesso da integração na camada administrativa, o estado espe-

rado em *devices*, no painel *Base View*, deve transitar de “*INSTALLING*” para “*UP*”, conforme apresentado na Figura 21.

Type	Hostname	Status	MAC	Category	IP	Network	Actions
Head Node	bcm11-headnode	Up	BC:24:11:56:49:F3		10.10.10.254	internal	[Icon] [Icon]
Head Node	master2	Up	BC:24:11:E3:C5:74		10.10.10.253	internal	[Icon] [Icon]
Physical Node	mg-site-director	Down	BC:24:11:EA:05:99	edge-director-mg-site	10.10.10.50	mg-site	[Icon] [Icon]
Physical Node	mg-site-node	Down	00:00:00:00:00:00	edge-mg-site-nodes	10.161.0.10	edge-tr	[Icon] [Icon]
Physical Node	node001	Up	BC:24:11:1D:6D:...	default	10.10.10.11	internal	[Icon] [Icon]
Physical Node	node002	Down	BC:24:11:6B:0D:...	default	10.10.10.12	internal	[Icon] [Icon]

Figura 21: Situação dos dispositivos incluídos.

Quando o *status* atinge a marcação UP, o nó de processamento é considerado integrado. A partir deste momento, qualquer atualização crítica de *drivers* ou *kernel* não é mais realizada máquina por máquina. Sendo assim, as atualizações são aplicadas apenas na *imagem* armazenada no *Head Node*, e todos os nós atrelados àquela categoria de imagem são sincronizados, e reiniciados automaticamente pelo BCM para refletir o novo estado. Esse modelo de gestão baseado em categorias reduz substancialmente o risco de desvio de configuração (*configuration drift*) entre os nós do *cluster*, já que todos os servidores de uma mesma categoria permanecem perpetuamente sincronizados com uma única fonte de verdade.

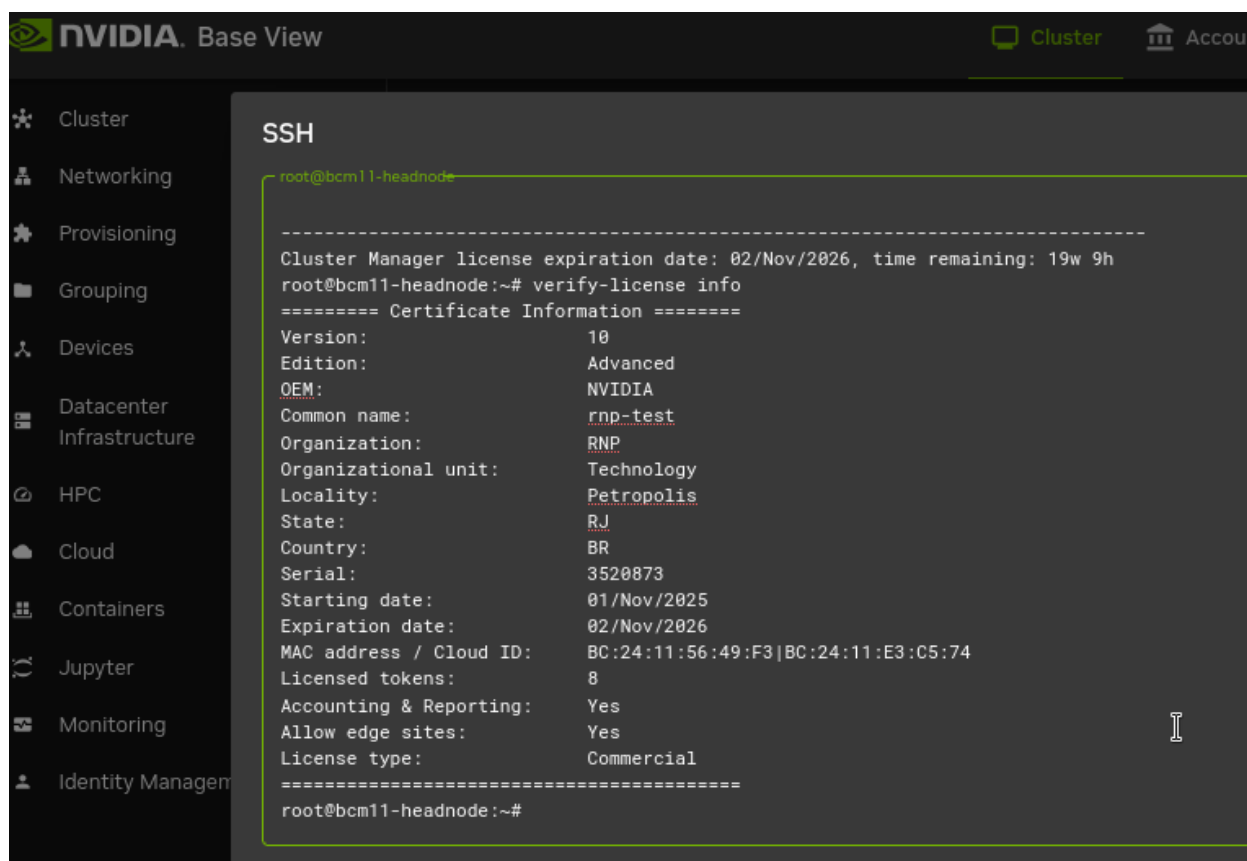
4.4.4 Extensão da Infraestrutura para a Borda (*Edge Computing*)

Embora o provisionamento de servidores locais no *data center* central cubra a maior parte das cargas de trabalho de IA, o Cenário C também avalia a arquitetura de borda, também conhecida como *Edge Computing*. Em ambientes onde a latência de rede ou a soberania dos dados exige que a inferência ou o pré-processamento ocorram fisicamente próximos à fonte geradora, como por exemplo em câmeras de linhas de montagem e sensores *Internet of Things* (IoT), o BCM permite estender o domínio do *cluster* remotamente de forma nativa e unificada. Essa capacidade elimina a necessidade de soluções de gerenciamento paralelas para os nós remotos, preservando as mesmas políticas de segurança e automação já aplicadas à infraestrutura central.

A arquitetura do BCM *Edge* baseia-se na figura do *Edge Director*, um servidor intermediário alocado na borda, que atua como um *proxy* de provisionamento e armazenamento de imagens. Essa intermediação é particularmente relevante em cenários de conectividade instável com o *data center* central, permitindo que o nó de borda continue operante mesmo diante de interrupções temporárias no enlace de rede. Dessa forma, o *Edge Director* atua como um espelho local da infraestrutura central, armazenando em *cache* as imagens de *software* e as configurações necessárias para que os nós remotos continuem sendo provisionados e mantidos mesmo durante períodos de indisponibilidade do enlace principal. Os passos a seguir descrevem a implantação de um nó de borda.

4.4.4.1 Ativação e Requisitos de Licenciamento

Diferentemente das funcionalidades nativas da rede local, a orquestração de borda é um módulo avançado do BCM. A primeira etapa do processo consistiu na validação e ativação da licença apropriada. Como o ambiente utiliza o licenciamento do ecossistema NVAIE, a ativação do módulo de *edge* requer que o *entitlement* (direito de uso) correspondente esteja registrado, como na Figura 22. Essa exigência reflete o modelo de licenciamento por módulos adotado pela suíte NVAIE, no qual funcionalidades avançadas, como a extensão para borda, são disponibilizadas de forma segmentada conforme o contrato firmado com a NVIDIA.



```

NVIDIA Base View
Cluster Account

Cluster
Networking
Provisioning
Grouping
Devices
Datacenter Infrastructure
HPC
Cloud
Containers
Jupyter
Monitoring
Identity Manager

SSH

root@bcm11-headnode:~#
-----
Cluster Manager license expiration date: 02/Nov/2026, time remaining: 19w 9h
root@bcm11-headnode:~# verify-license info
===== Certificate Information =====
Version: 10
Edition: Advanced
OEM: NVIDIA
Common name: rnp-test
Organization: RNP
Organizational unit: Technology
Locality: Petropolis
State: RJ
Country: BR
Serial: 3520873
Starting date: 01/Nov/2025
Expiration date: 02/Nov/2026
MAC address / Cloud ID: BC:24:11:56:49:F3|BC:24:11:E3:C5:74
Licensed tokens: 8
Accounting & Reporting: Yes
Allow edge sites: Yes
License type: Commercial
=====
root@bcm11-headnode:~#

```

Figura 22: Informações sobre a licença.

A atualização da licença pode ser aplicada diretamente pela *Base View* ou através do *shell* administrativo (*cmsh*). Uma vez que o *Head Node* valida o certificado de licenciamento, os módulos de rede, túneis seguros e computação de borda, são desbloqueados.

4.4.4.2 Implantação do *Edge Director* via *Script*

A partir do nó principal, a criação do perfil e da imagem base do *Edge Director* é orquestrada pelo comando interativo *cm-edge-setup*. Este utilitário guia o administrador pelas seguintes definições:

- Escolha do protocolo de túnel seguro (como o WireGuard e o OpenVPN) e portas de comunicação expostas na rede pública, a *Wide Area Network* (WAN);
- Definição do bloco *Classless Inter-Domain Routing* (CIDR) para a rede de borda;

- Geração de uma imagem de disco para inicialização e provisionamento remoto do servidor que assumirá o papel de diretor do ambiente de borda.

A imagem de disco gerada pelo comando `cm-edge-setup` é então copiada para o repositório de imagens do Proxmox e a máquina virtual do *Edge Director* criada e configurada para inicialização a partir desta imagem, cujo processo de instalação e visão final do dispositivo ativo podem ser observados nas Figuras 23 e 24, respectivamente.

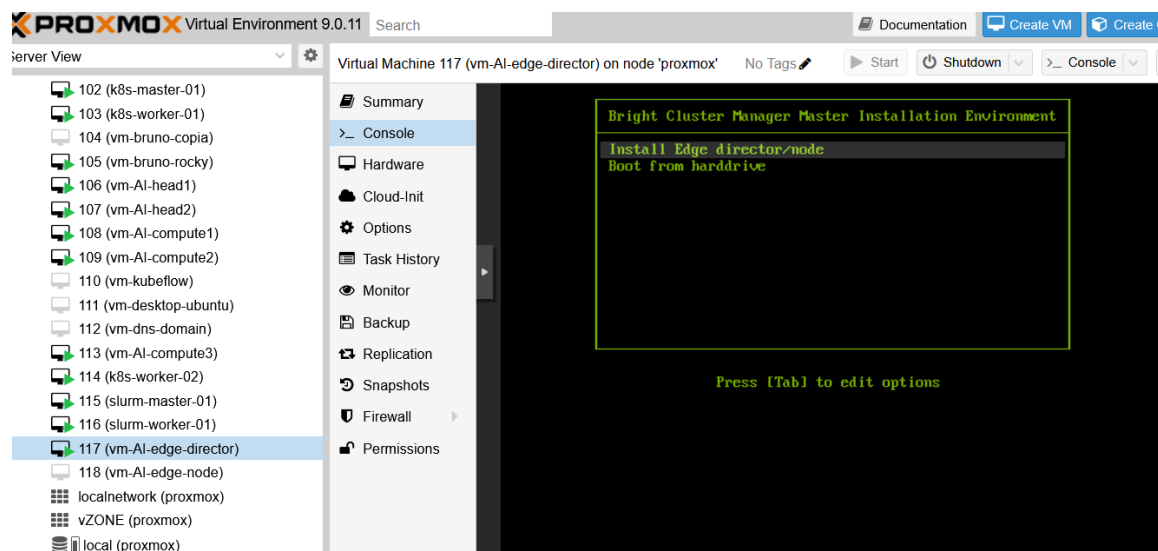


Figura 23: Exemplo de instalação do *Edge Director*.

☐	Head Node	master2	↑ Up	BC:24:11:E3:C5:74	10.10.10.253	internalnet	
☐	Physical Node	mg-site-director	↑ Up	BC:24:11:EA:05:99	edge-director-mg-site	10.10.10.50	mg-site-external mg-edge-director
☐	Physical Node	mg-site-node	↓ Down	00:00:00:00:00:00	edge-mg-site-nodes	10.161.0.10	edge-test-internal mg-edge-node

Figura 24: Visão do dispositivo de borda ativo.

4.4.4.3 Inclusão e Provisionamento do *Edge Node*

Com o *Edge Director* instalado em uma rede virtual distinta simulando um local remoto, o diretor passa a fornecer os serviços críticos locais para a rede da filial. A inclusão de um *Edge Node* segue de forma semelhante ao processo descrito anteriormente.

Essa arquitetura elimina a necessidade de equipe de TI especializada em todas as filiais. Uma vez que o servidor é energizado e conectado à rede na borda, todo o fatiamento de armazenamento, distribuição de *drivers* NVIDIA e integração com o orquestrador (Kubernetes ou Slurm) ocorre de forma invisível, administrado a partir da console central.

4.4.5 Particionamento Físico de Aceleradores (MIG)

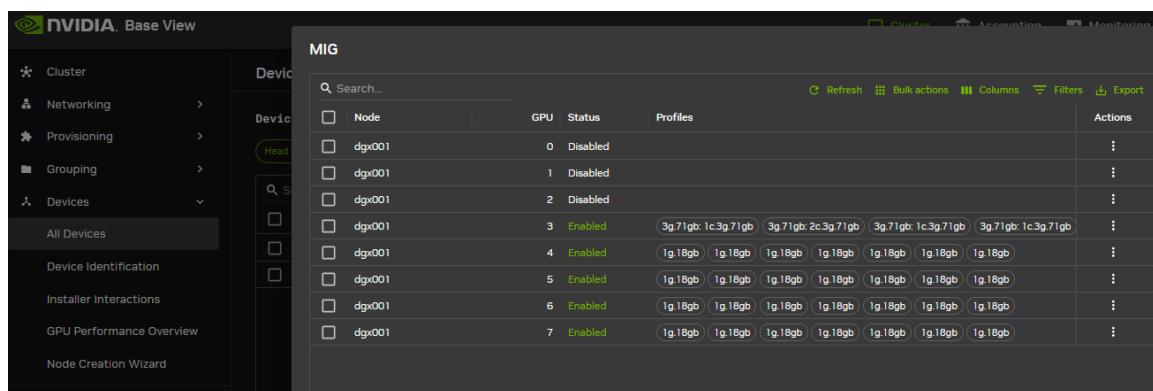
Enquanto o ecossistema *cloud-native* detalhado no Cenário B utiliza abordagens baseadas em *software*, como por exemplo o *Volcano Device Plugin*, para fatiar virtualmente a memória de vídeo, já que as GPUs não possuem o recurso MIG, no Cenário C aproveita-se do particionamento direto no silício através desta tecnologia MIG, compatível com as GPUs de arquitetura *Ampere* ou *Hopper*. Os perfis MIG relacionados à GPU utilizada (H200) permitem dividir uma

única GPU física em até sete instâncias isoladas, cada uma operando com sua própria largura de banda, alocação de *cache* L2 e motores de computação independentes [52]. Diferentemente do fatiamento por *software*, essa segmentação em nível de *hardware* garante isolamento de falhas entre as instâncias, de modo que um processo com comportamento anômalo em uma partição não compromete o desempenho ou a estabilidade das demais instâncias na mesma placa física.

A gestão dessa topologia pode ser operacionalmente realizada via linha de comando, através do utilitário `nvidia-smi`. No entanto, o BCM simplifica esse fluxo ao centralizar a configuração diretamente na interface *Base View*, eliminando a necessidade de acesso individual via *shell* a cada nó equipado com GPUs para a definição dos perfis MIG desejados.

4.4.5.1 Configuração Heterogênea do Nó (Exemplo Prático)

Conforme ilustrado na Figura 25, o painel de administração de dispositivos apresentado através do caminho *Devices > All Devices > MIG* permite que o administrador defina o perfil de execução de cada acelerador individualmente.



The screenshot shows the NVIDIA Base View interface for configuring MIG profiles. The left sidebar contains navigation options like Cluster, Networking, Provisioning, Grouping, and Devices. The main panel displays a table of MIG profiles for a specific node.

Node	GPU	Status	Profiles	Actions
dgx001	0	Disabled		
dgx001	1	Disabled		
dgx001	2	Disabled		
dgx001	3	Enabled	3g.71gb: 1c.3g.71gb 3g.71gb: 2c.3g.71gb 3g.71gb: 1c.3g.71gb 3g.71gb: 1c.3g.71gb	
dgx001	4	Enabled	1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb	
dgx001	5	Enabled	1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb	
dgx001	6	Enabled	1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb	
dgx001	7	Enabled	1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb 1g.18gb	

Figura 25: Perfis MIG criados em GPU DGX H200.

A captura de tela demonstra a configuração estratégica de um nó, evidenciando um ambiente para atender a múltiplas demandas do projeto, tais como:

- GPUs 0, 1 e 2 (MIG *Disabled*): esta configuração é reservada para cargas de trabalho monolíticas, como treinamentos distribuídos de *Large Language Model* (LLM) ou processamento paralelo massivo, nos quais a aplicação precisa de acesso irrestrito a toda a memória e aos *links* de interconexão direta entre as GPUs (NVLink) [35];
- GPU 3 (Perfis Intermediários): o MIG está habilitado e configurado com fatias médias, balanceando o isolamento e a oferta de recursos para inferências mais pesadas ou sessões interativas de desenvolvimento que exigem mais memória;
- GPUs 4, 5, 6 e 7 (Fatiamento de Alta Densidade): estão configuradas com perfis de extrema granularidade, com sete instâncias isoladas do tipo 1g por placa. Esta topologia otimiza radicalmente a taxa de utilização do *hardware*, permitindo que até 28 pesquisadores ou serviços de *JupyterHub* ou *Kubernetes* executem cargas de trabalho leves, como depuração de código ou inferência básica simultaneamente no mesmo servidor, sem qualquer risco de contenção de barramento ou interferência mútua.

4.4.6 Implantação Automatizada do Kubernetes via BCM

Em forte contraste com a implantação manual e fragmentada via `kubeadm` detalhada no Cenário B, a orquestração de contêineres na arquitetura corporativa é tratada como um serviço nativo da infraestrutura. O BCM abstrai completamente a complexidade de instalação, geração de certificados, configuração de chaves e integração de rede através do seu assistente de configuração unificado [53], conforme ilustrado na Figura 26.

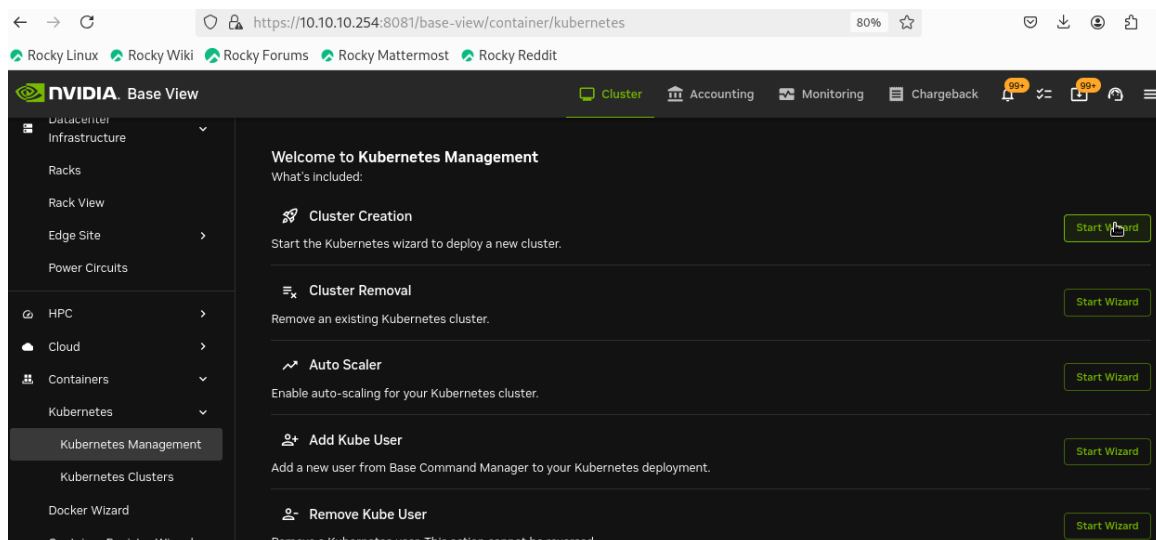


Figura 26: Interface do K8S *deploy Wizard*.

Para consolidar o ecossistema *cloud-native* descrito no Cenário C, é necessário provisionar a malha do orquestrador Kubernetes. Diferente da instalação manual do Cenário B, este procedimento é automatizado pelo BCM através de uma interface baseada em papéis (*roles*). As etapas a seguir descrevem a sequência operacional adotada pela equipe técnica para instanciar o K8S e vincular os nós de processamento à malha do *cluster*.

4.4.6.1 Inicialização do Assistente

A implantação do ecossistema *cloud-native* pode ser conduzida tanto pela interface gráfica (*Base View*) quanto pelo terminal administrativo. O processo inicia-se invocando o utilitário de configuração, que audita o estado atual da rede interna e prepara os pacotes binários necessários para distribuição:

```
cm - kubernetes - setup
```

4.4.6.2 Definição de Topologia e Papéis

Diferentemente da alocação manual, o BCM adota um paradigma baseado na atribuição de papéis arquitetônicos. Durante o assistente, o administrador mapeia as seguintes responsabilidades sem precisar acessar as máquinas individualmente:

- *Control Plane*: geralmente atribuído ao nó principal ou a um conjunto de nós de gerência. O BCM instala e configura componentes como o `kube-apiserver`, `etcd` e `kube-scheduler`.
- *workers*: atribuído dinamicamente às categorias de nós de processamento. Ao associar o papel de *Worker* ao servidor DGX [54] da topologia, o BCM garante que o agente

kubelet e o motor de contêineres sejam injetados e configurados para iniciar junto ao sistema operacional do nó.

4.4.6.3 Configuração de Rede e Armazenamento

O assistente elimina a necessidade de aplicar manifestos YAML externos para estabelecer a malha de comunicação. O BCM provisiona nativamente soluções de CNI. Neste cenário, optou-se pela adoção do Flannel devido à sua simplicidade arquitetural, ou seja, uma rede de sobreposição (*overlay network*) leve que configura uma malha de roteamento de camada 3, alocando uma sub-rede para cada nó e permitindo a comunicação transparente entre os Pods em todo o *cluster* [55].

4.4.6.4 Integração Nativa com o Ecossistema NVIDIA (GPU Operator)

Ao detectar que os *workers* selecionados possuem *hardware* NVIDIA e partições MIG ativas conforme a topologia heterogênea configurada no passo anterior, o assistente do BCM implanta automaticamente o NVIDIA GPU Operator no *cluster*, caso o mesmo seja selecionado no *wizard*, conforme ilustrado na Figura 27.

```
root@bcm11-headnode:~# kubectl get all -n gpu-operator
```

NAME	READY	STATUS	RESTARTS	AGE
pod/gpu-feature-discovery-ntp5	1/1	Running	0	69d
pod/gpu-operator-756f8b499-p7wvj	1/1	Running	28 (24d ago)	97d
pod/gpu-operator-node-feature-discovery-gc-55ffc49ccc-dp2z1	1/1	Running	0	97d
pod/gpu-operator-node-feature-discovery-master-6b5787f695-z6c8k	1/1	Running	22 (24d ago)	97d
pod/gpu-operator-node-feature-discovery-worker-hgmjp	1/1	Running	8 (37d ago)	97d
pod/gpu-operator-node-feature-discovery-worker-n2qw2	1/1	Running	21 (69d ago)	97d
pod/nvidia-container-toolkit-daemonset-p7dp4	1/1	Running	0	97d
pod/nvidia-cuda-validator-8kl72	0/1	Completed	0	69d
pod/nvidia-dcgm-exporter-jzvix	1/1	Running	0	97d
pod/nvidia-device-plugin-daemonset-vmcw7	1/1	Running	0	69d
pod/nvidia-mig-manager-rgrtq	1/1	Running	0	97d
pod/nvidia-operator-validator-zq5q2	1/1	Running	0	69d

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/gpu-operator	ClusterIP	10.150.219.25	<none>	8888/TCP	97d
service/nvidia-dcgm-exporter	ClusterIP	10.150.86.63	<none>	9400/TCP	97d

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR	AGE
daemonset.apps/gpu-feature-discovery	1	1	1	1	1	nvidia.com/gpu.deploy.gpu-feature-discovery=true	97d
daemonset.apps/gpu-operator-node-feature-discovery-worker	2	2	2	2	2	<none>	97d
daemonset.apps/nvidia-container-toolkit-daemonset	1	1	1	1	1	nvidia.com/gpu.deploy.container-toolkit=true	97d
daemonset.apps/nvidia-dcgm-exporter	1	1	1	1	1	nvidia.com/gpu.deploy.dcg-exporter=true	97d
daemonset.apps/nvidia-device-plugin-daemonset	1	1	1	1	1	nvidia.com/gpu.deploy.device-plugin=true	97d
daemonset.apps/nvidia-device-plugin-mps-control-daemon	0	0	0	0	0	nvidia.com/gpu.deploy.device-plugin=true,nvidia.com/mps.capable=true	97d
daemonset.apps/nvidia-mig-manager	1	1	1	1	1	nvidia.com/gpu.deploy.mig-manager=true	97d
daemonset.apps/nvidia-operator-validator	1	1	1	1	1	nvidia.com/gpu.deploy.operator-validator=true	97d

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/gpu-operator	1/1	1	1	97d
deployment.apps/gpu-operator-node-feature-discovery-gc	1/1	1	1	97d
deployment.apps/gpu-operator-node-feature-discovery-master	1/1	1	1	97d

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/gpu-operator-5fcd9d598	0	0	0	97d
replicaset.apps/gpu-operator-756f8b499	1	1	1	97d
replicaset.apps/gpu-operator-node-feature-discovery-gc-55ffc49ccc	1	1	1	97d
replicaset.apps/gpu-operator-node-feature-discovery-master-646f68958c	0	0	0	97d
replicaset.apps/gpu-operator-node-feature-discovery-master-6b5787f695	1	1	1	97d

```
root@bcm11-headnode:~#
```

Figura 27: GPU Operator e componentes em execução no *cluster*.

O GPU Operator atua como o controlador definitivo do ciclo de vida das placas de vídeo dentro do Kubernetes. Ele substitui a necessidade de configurações manuais fragmentadas, implantando automaticamente os *drivers*, o NVIDIA *container Toolkit*, o NVIDIA *Device Plugin* e os exportadores de métricas para monitoramento. Ele rastreia instantaneamente as fatias de *hardware* isoladas pelo MIG e as registra na API do orquestrador como recursos alocáveis, deixando a infraestrutura perfeitamente homologada para receber as plataformas de MLOps subsequentes.

4.4.7 Portal Interativo de Desenvolvimento (JupyterHub)

A facilidade operacional da arquitetura corporativa estende-se além do plano de controle do Kubernetes, alcançando as ferramentas de produtividade destinadas ao usuário final. No Cenário C, a disponibilização do JupyterHub abdicou inteiramente do gerenciamento manual de manifestos ou do isolamento de diretórios via linhas de comando complexas do Helm. Em vez

disso, a plataforma foi instanciada de forma centralizada utilizando o assistente de configuração unificado do BCM (*JupyterHub Wizard*), conforme mostrado na Figura 28.

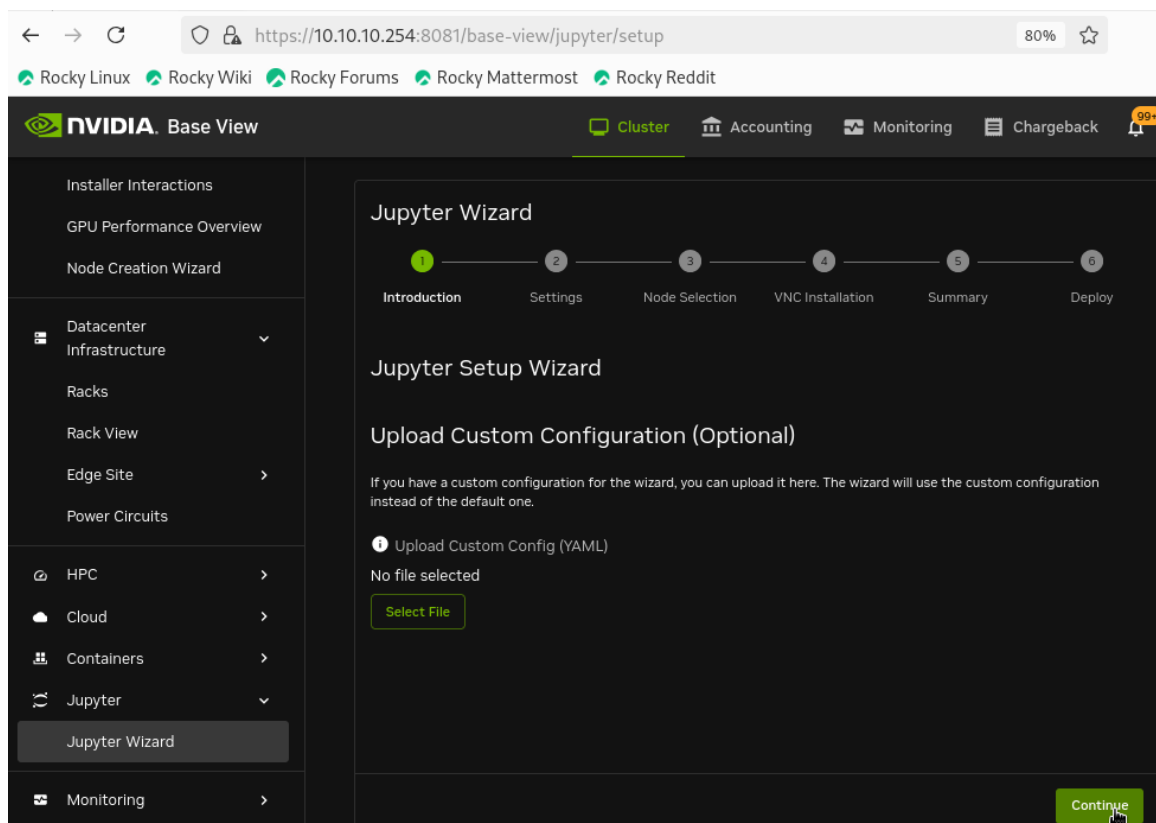


Figura 28: Interface gráfica para *deploy* do JupyterHub.

Essa abordagem reflete a mesma filosofia de convergência aplicada às demais camadas do Cenário C, na qual a complexidade inerente à integração entre autenticação, armazenamento persistente e orquestração de contêineres é abstraída em um fluxo guiado, acessível mesmo a administradores sem profundo conhecimento do ecossistema Kubernetes.

Além disso, o processo garante que o portal seja criado em total conformidade com a infraestrutura *cloud-native* e as diretrizes de segurança de rede estabelecidas pelo BCM, bem como integrado com o Slurm e K8S. Como consequência direta dessa integração nativa, qualquer usuário provisionado através do portal já herda automaticamente as políticas de acesso e as permissões de rede definidas centralmente, dispensando etapas adicionais de configuração manual pós-implantação.

4.4.7.1 Execução do Assistente de Instalação

O provisionamento da plataforma é disparado a partir do *Base View*, através de um assistente que audita o ambiente e prepara o ecossistema do JupyterHub de forma totalmente guiada. Durante a execução do *script*, o administrador parametriza os aspectos operacionais críticos através de menus simplificados, incluindo a definição do método de autenticação dos pesquisadores e o provisionamento de armazenamento persistente para as pastas *home* dos usuários. Essa abordagem assistida reduz significativamente a chance de erros de configuração manual, dispensando o conhecimento aprofundado dos manifestos YAML e comandos subjacentes ao Helm que seriam necessários em uma implantação puramente declarativa. Para o

acesso à interface de login gerada, é preciso que as contas façam parte do grupo *user portal* presente na seção de identidades do BCM.

4.5 Resultados Consolidados

A execução das POCs estabelecidas ao longo deste estudo permitiu uma avaliação empírica e comparativa entre três paradigmas distintos de orquestração de infraestrutura para IA e HPC. O contraste entre as abordagens revelou *trade-offs* cruciais entre flexibilidade e complexidade operacional, evidenciando que a escolha ideal depende diretamente do perfil de carga de trabalho e da maturidade operacional da equipe responsável. A seguir, é descrita uma visão geral dos cenários avaliados.

4.5.1 Síntese dos Cenários Avaliados

- Cenário A (HPC tradicional): a arquitetura baseada em servidores *bare-metal* e Slurm demonstrou extrema robustez e baixa latência intrínseca. Contudo, o engessamento na gestão de dependências, a incapacidade de fatiar GPUs virtualmente e a falta de alinhamento com os padrões *cloud-native* (estabelecidos pela CNCF) tornam este modelo restritivo. Esse distanciamento tecnológico dificulta a adoção de esteiras CI/CD, tornando a arquitetura ineficiente tanto para a entrega ágil de aplicações modernas quanto para fluxos de trabalho interativos de prototipagem em IA, onde a ociosidade de recursos é frequente.
- Cenário B (*Cloud-Native* / Fatiamento Virtual): a transição para o ecossistema Kubernetes, orquestrado sobre o Proxmox, provou que é possível modernizar a infraestrutura de HPC tradicional. O escalonador Volcano introduziu o fatiamento virtual de GPUs, e o projeto Slinky [22] validou a viabilidade de rodar o HPC clássico de forma efêmera e containerizada. A lição aprendida, no entanto, é o elevado *overhead* de engenharia. Em outras palavras, a sustentação desta malha de ferramentas *open-source* exige um esforço de integração e planejamento.
- Cenário C (NVAIE): a arquitetura de referência corporativa mitigou o abismo operacional dos cenários anteriores, pois o provisionamento automatizado de imagens via BCM eliminou a configuração fragmentada. Ao centralizar a orquestração em uma pilha homologada, houve redução significativa do tempo de configuração do ambiente para os experimentos. Por fim, a adoção dessa solução integrada consolidou a convergência necessária entre HPC e IA, entregando uma interface unificada que viabiliza a construção e treinamento de modelos em um ambiente robusto.

5 *Status* Atual e Próximos Passos

Durante o período de aquisição dos equipamentos que compõem a infraestrutura, foi possível trilhar os caminhos descritos nas seções anteriores deste relatório. Essa etapa preliminar foi fundamental para compreender, de maneira antecipada e estratégica, quais detalhes técnicos e logísticos seriam importantes para o sucesso da instalação física e lógica dos ativos.

Nesta seção, apresentamos um panorama detalhado sobre o momento atual do ambiente de computação de alto desempenho, consolidando o *status* das entregas e das configurações já realizadas, visando a homologação final e a disponibilização do ambiente para a comunidade usuária.

5.1 Visão Geral do *Status* prévio

Inicialmente foi realizado um mapeamento dos requisitos técnicos e operacionais do HPC, sendo estes definidos com base nas demandas e fluxos de trabalho dos pesquisadores vinculados ao projeto, conforme detalhado em [1]. Essa etapa de diagnóstico foi essencial para traduzir as necessidades científicas e de desenvolvimento em especificações de engenharia.

A partir desse levantamento, foi possível determinar as características essenciais que os equipamentos deveriam possuir, garantindo que a infraestrutura estivesse perfeitamente alinhada aos objetivos do projeto. Adotou-se então como arquitetura de referência a solução computacional denominada NVIDIA BasePOD, representada na Figura 29, composta pelos seguintes elementos:

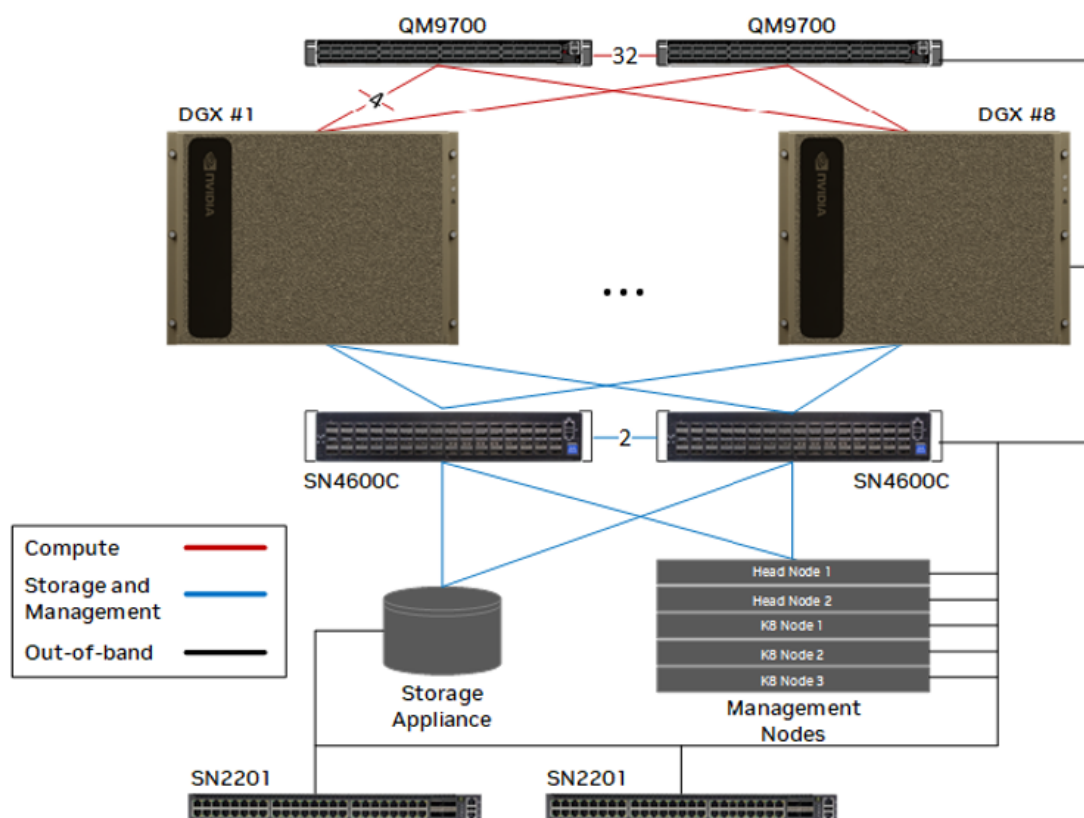


Figura 29: NVIDIA BasePOD: Arquitetura de referência.

- Camada de Processamento: formada por servidores com GPUs, iniciando em configurações com 2 nós e escalando de forma modular;
- Camada de Controle: formada por servidores de gerenciamento e orquestração do ambiente;
- Camada de Rede de Computação: rede dedicada à interconexão de GPUs de diferentes servidores. Faz com que o *cluster* atue como um único supercomputador, viabilizando o treinamento de IA em larga escala;
- Camada de Rede de Armazenamento e Gerenciamento: rede focada no tráfego de dados com a solução de armazenamento e utilizada também na gerência da infraestrutura;
- Camada de Armazenamento: baseada em arquiteturas validadas pela NVIDIA em parceria com fornecedores certificados, garantindo altíssima taxa de transferência e baixíssima latência.

5.1.1 Equipamentos Adquiridos

Após avaliação das propostas recebidas, no final do mês de junho de 2025 o processo de aquisição da infraestrutura teve a formalização de compra dos equipamentos junto ao fornecedor selecionado. A conclusão desta etapa administrativa e jurídica validou o atendimento a todas as exigências técnicas estipuladas no Termo de Referência (TR).

Ao identificar uma particularidade do ambiente do projeto, optou-se por não adquirir equipamentos dedicados para a Camada de Rede de Computação, pois após mapear junto aos pesquisadores os casos de uso do ambiente computacional, identificamos que neste momento inicial não haveria fluxos de processamento que demandem mais do que oito GPUs, ou seja, um nó computacional. Sendo assim, a solução computacional ilustrada na Figura 30 é composta de quatro partes:

- Nós de processamento em GPU, identificados com o número 1;
- Nó de armazenamento de dados, identificado com o número 2;
- Nós de controle e Kubernetes, identificados com o número 3;
- Equipamentos de interconexão de redes, identificados com o número 4.

A seguir, são detalhadas as características técnicas de cada um dos nós da arquitetura proposta para a solução computacional:

- Nós de processamento em GPU: 2 servidores HPE Cray XD670 com a seguinte configuração: 2 processadores Intel Xeon Platinum 8480+ 2,0 GHz 56C/112T, 8 GPUs NVIDIA H200 141 GB, 2 TB de memória RAM, 30 TB de armazenamento, 8 interfaces de rede 400 *Gigabits Per Second* (Gbps) e 4 interfaces de rede 200 Gbps;
- Nó de armazenamento de dados: 1 sistema de armazenamento paralelo HPE Cray C500, baseado no sistema de arquivos Lustre, com 4 nós e 368 TB de armazenamento;
- Nós de controle e Kubernetes: 4 servidores HPE DL385 Gen11 compostos com 2 processadores AMD EPYC 9334 2,7 GHz 32C/64T, 512 GB ou 256 GB de memória RAM, 3.8 TB de armazenamento, e 4 interfaces de rede de 200 Gbps;

- Equipamentos de interconexão de redes: 2 switches HPE SN3700M 200 Gbps, e 1 switch HPE SN3420M 25 Gbps.

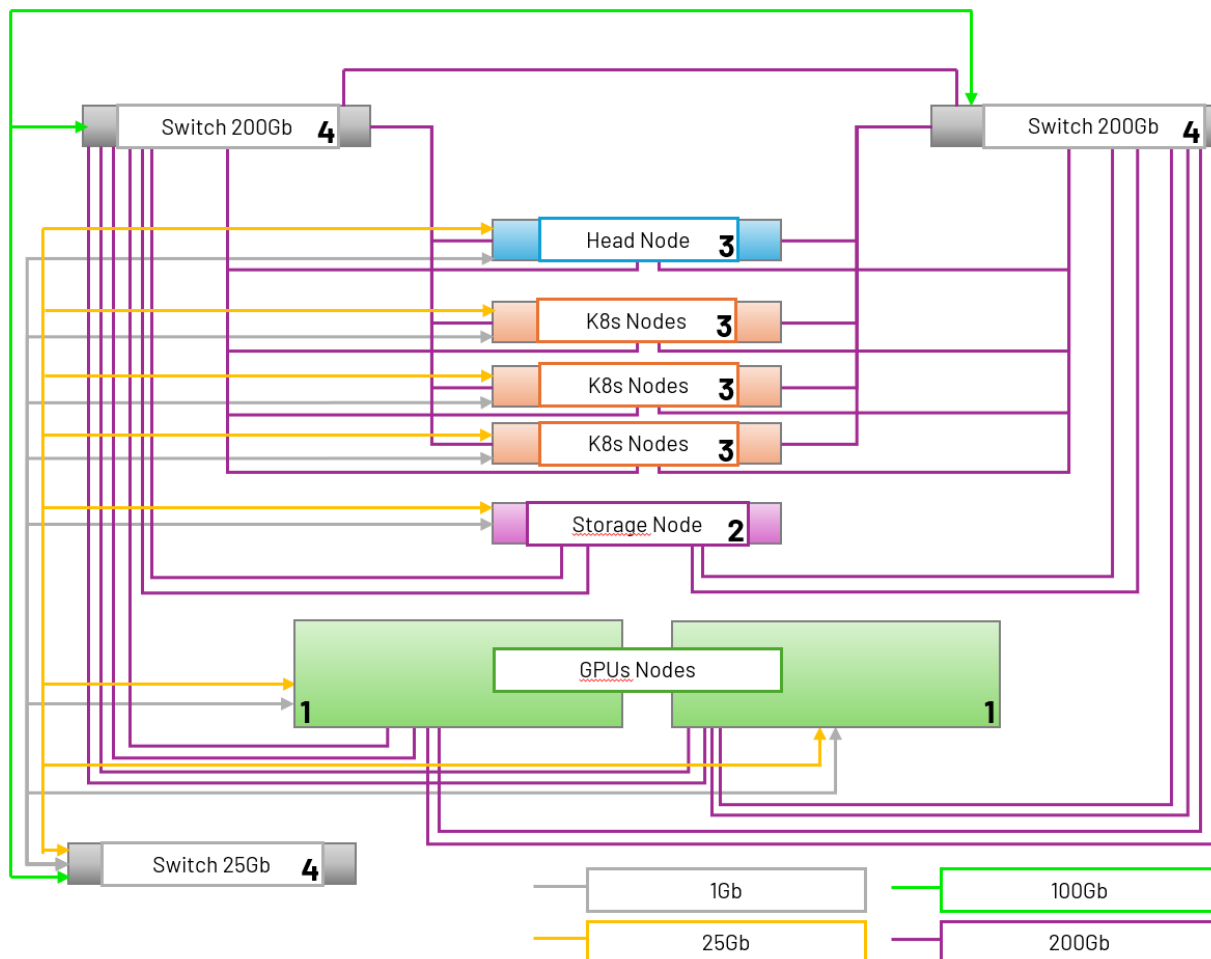


Figura 30: Arquitetura da solução computacional.

5.1.2 Cotas de Importação e os Entraves da Ciência Brasileira

A Lei nº 8.010/1990 [56] instituiu o regime de isenção fiscal para a importação de bens destinados à pesquisa científica e tecnológica, mediante uma cota anual gerenciada pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq). Em razão do esgotamento da cota correspondente ao exercício de 2025, o processo de importação dos referidos equipamentos foi temporariamente suspenso, aguardando a renovação do teto de isenção com a abertura do calendário de 2026.

A deterioração severa do teto autorizado para importações científicas isentas de impostos, que sofreu uma redução drástica de 400 milhões de dólares em 2023 para apenas 229 milhões de dólares em 2025 [57], tem provocado o esgotamento precoce dessas cotas logo nos primeiros meses do ano. Essa limitação gera uma profunda contradição orçamentária, pois ocorre no mesmo período em que os investimentos públicos gerais do setor expandiram. O Fundo Nacional de Desenvolvimento Científico e Tecnológico (FNDCT) operando em cerca de 17,7 bilhões de reais anuais [58], mas a capacidade material de adquirir insumos estrangeiros essenciais permanece

contida. Como consequência direta, projetos de pesquisa em andamento são paralisados, a formação de novos pesquisadores é afetada e a soberania científica do país é severamente comprometida, prejudicando áreas estratégicas da ciência brasileira.

5.1.3 Local de Instalação e Entrega das Máquinas

Um requisito fundamental para o local de instalação é sua integração com a infraestrutura da Rede Nacional de Ensino e Pesquisa (RNP), integrando um dos Centro Nacional de Dados (CNDs) da instituição. Esses CNDs estão estrategicamente distribuídos no Brasil dentro de *data centers* parceiros com certificação Tier III, que hospedam serviços críticos e a infraestrutura central da rede.

Em janeiro de 2026, foi realizada vistoria técnica no CND Brasília que atestou as condições apropriadas de refrigeração e energia, destacando-se o sistema de confinamento de corredores (ar quente e ar frio), que otimiza a eficiência térmica e proporciona as temperaturas baixas exigidas para a operação de equipamentos de alta densidade calorífica. Durante a inspeção, validaram-se aspectos técnicos estruturais essenciais para o funcionamento dos ativos adquiridos, incluindo a profundidade dos racks e das calhas de cabeamento, bem como a quantidade de racks disponíveis para expansão. Adicionalmente, foram inspecionadas as instalações do sistema de geração de energia de emergência por grupos geradores e as centrais de água gelada e chillers responsáveis pela refrigeração ambiental.

Tal vistoria se junta à vistoria técnica realizada no CND São Paulo em 2024, a qual constatou a inviabilidade do local para o recebimento de uma infraestrutura computacional com requisitos rígidos, devido à capacidade insuficiente de refrigeração de equipamentos que gerem uma grande quantidade de calor.

Em fevereiro de 2026, os equipamentos desembarcaram no Brasil e iniciaram o processo burocrático de nacionalização, sendo de fato entregues no CND Brasília somente no começo do mês de março de 2026.

5.2 Visão do *Status* Atual de Implantação do HPC

A instalação física de todos os equipamentos foi concluída no final de março de 2026, exigindo adequações de última hora nas *Power Distribution Units* (PDUs) do *data center* do CND devido à especificação dos cabos de alimentação dos nós de processamento equipados com GPU. Esses nós possuem seis fontes de alimentação de 3.600 W cada, operando na faixa de 240 V a 277 V.

Para suportar a infraestrutura adquirida, foi implantado um par redundante de *Firewalls* Fortinet, garantindo a segurança de perímetro da solução. Esses ativos conectam-se a dois switches Cisco NCS 540 (SW-SPINE-CND-1 e SW-SPINE-CND-2), que realizam enlances de 100 Gbps com os switches HPE SN3700M da infraestrutura computacional, os quais, por sua vez, interconectam-se ao switch SN3420 (SW-BRASIL6G-3), completando a rota de comunicação externa com o *head node*. O diagrama desta estrutura está representado na Figura 31.

A interconexão entre *Head Node*, *Kubernetes Nodes*, *GPU Nodes* e *Storage Node* é realizada por uma rede de 200 Gbps, vide Figura 32, elemento central de integração do *cluster*, garantindo baixa latência entre as GPUs e os *datasets* armazenados. Essa largura de banda dedicada é essencial para evitar gargalos durante o treinamento distribuído de modelos, especialmente nas etapas de sincronização de gradientes entre múltiplos nós de GPU.

Em conformidade com a arquitetura de referência da NVIDIA, todos os nós de processamento foram interconectados em uma rede local privada e isolada, blindando os componentes

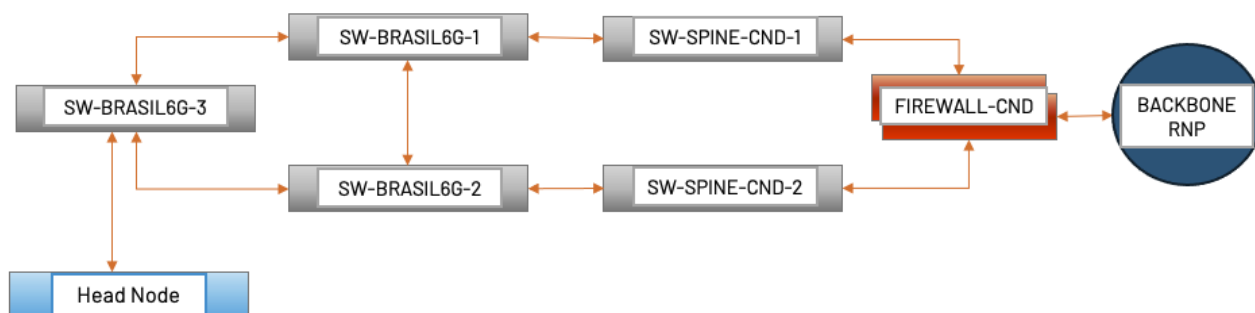


Figura 31: Rede externa conectada ao *Head Node* a 100 Gbps.

críticos contra varreduras e ataques externos. Nessa estrutura, o *head node* é o único elemento exposto à rede externa, com IP público, funcionando como ponto de entrada unificado para autenticação e controle de acesso de pesquisadores e usuários.

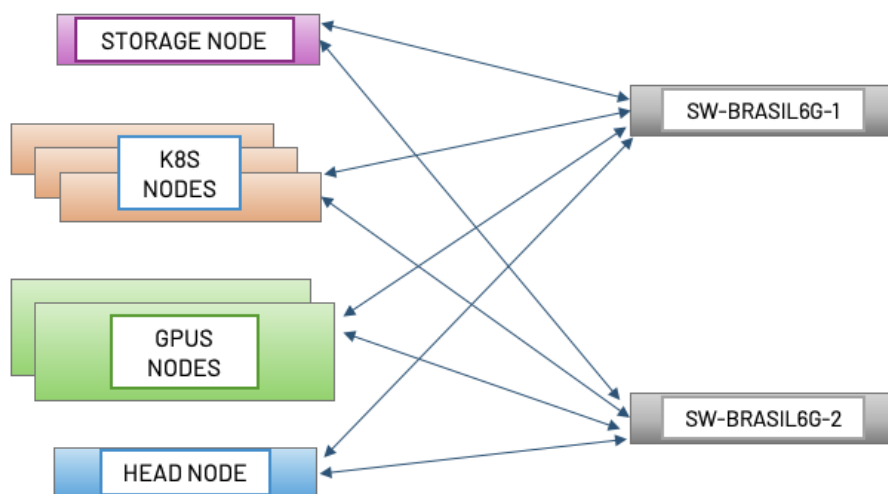


Figura 32: Rede interna interconectando todos os nós a 200 Gbps.

Diante do elevado consumo energético, que ultrapassa os limites de um único rack, os ativos foram distribuídos em dois racks contíguos, registrados nas Figuras 33 e 34, distribuindo as cargas térmica e energética. Um terceiro rack dedicado foi integrado para os equipamentos de segurança e conectividade com a rede da RNP.

5.2.1 Instalação da Aplicação

A plataforma computacional implantada é composta por um ecossistema de soluções modulares e interdependentes, projetadas para atuar de forma conjunta. A arquitetura de *software* divide-se logicamente entre a camada de infraestrutura de base e a camada de orquestração e gerenciamento de cargas de trabalho.

1. Pilares de Infraestrutura de Base



Figura 33: Visão frontal dos racks em processo de montagem - 1 à esquerda e 2 à direita.

O alicerce do *cluster* é sustentado por dois componentes fundamentais, responsáveis pelo provisionamento lógico e pela entrega de dados em altíssima velocidade:

- a) *Armazenamento Paralelo Baseado em Lustre*: Esta infraestrutura de *storage* foi projetada especificamente para atender às rigorosas demandas de sistemas HPC. Sua função principal é consolidar o armazenamento centralizado de conjuntos de dados (*datasets*), pesos de modelos de inteligência artificial, arquivos temporários e dados de saída (*outputs*) de múltiplos usuários. Para viabilizar o treinamento eficiente de redes neurais profundas pelas GPUs, o sistema de arquivos Lustre entrega *throughput* massivo e resiliência contra falhas através de redundância de *hardware* e *software*, eliminando gargalos de I/O (Entrada/Saída).
- b) *Base Command Manager*: Atuando como o “sistema operacional” do *cluster*, o BCM é a ferramenta central de gerenciamento de *hardware* e provisionamento de baixo nível. Ele abstrai a complexidade operacional da infraestrutura, permitindo o inventário, monitoramento de saúde e provisionamento automatizado de nós heterogêneos (nós de processamento GPU, nós de CPU e head nodes) a partir de uma interface unificada de controle, conforme ilustrado na Figura 35.

2. Camada de Orquestração e Agendamento

Em um nível superior de abstração, provisionados de forma integrada pelo BCM, operam ambos os motores de execução científica da plataforma, dimensionados para diferentes perfis de desenvolvimento:

- a) *Slurm Workload Manager*: De forma complementar à execução de contêineres em tempo real, o *Slurm* atua como o agendador de tarefas tradicional de HPC. Ele é responsável pelo gerenciamento de tarefas de processamento massivo e de longa duração. Os pesquisadores utilizam o *Slurm* para submeter *scripts* de execução não interativos a filas de processamento configuradas por políticas de prioridade

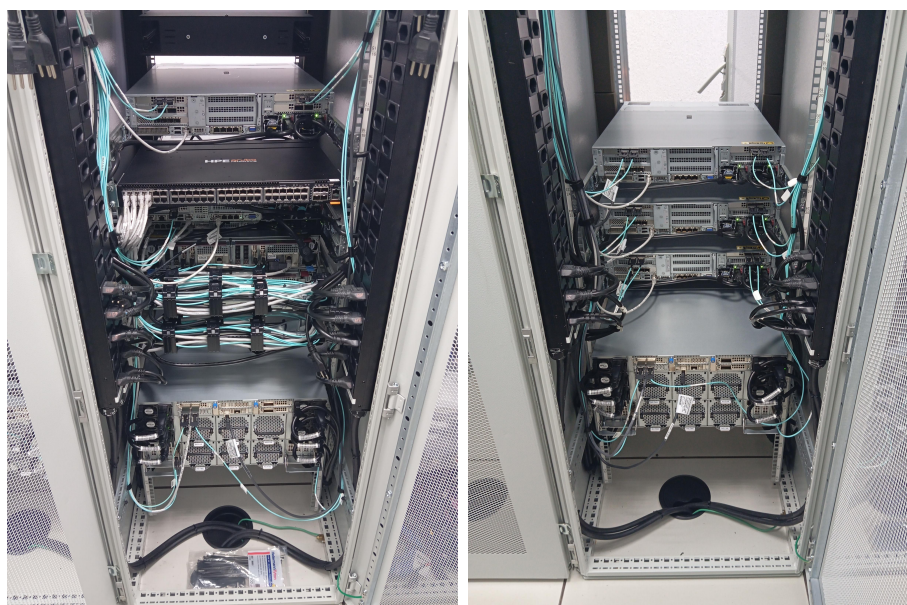


Figura 34: Visão traseira dos racks em processo de montagem - 1 à esquerda e 2 à direita.

e cota, garantindo que grandes simulações e treinamentos pesados distribuídos em múltiplos nós sejam escalonados de forma justa, segura e resiliente. A Figura 36 resume o conjunto de ferramentas disponibilizadas aos pesquisadores nesta camada.

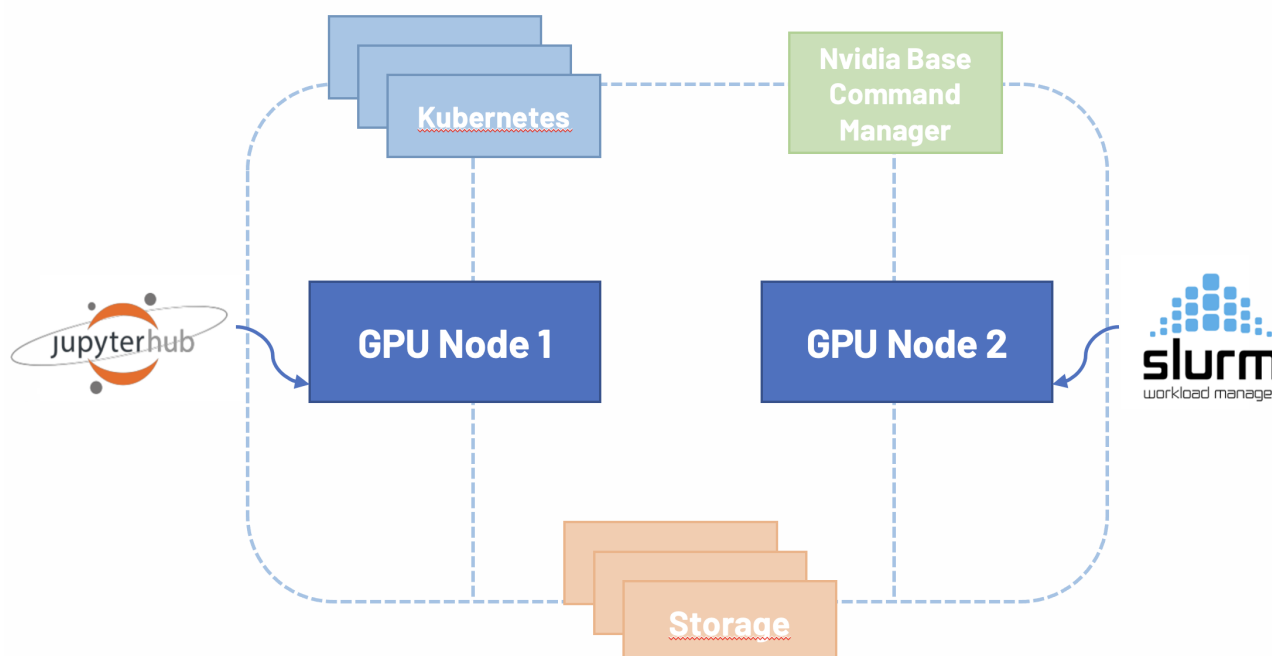


Figura 36: Ferramentas que serão disponibilizadas para os pesquisadores.

- b) Plataforma Run:ai: Recentemente integrada ao portfólio oficial de *software* da NVIDIA, a solução Run:ai funciona como uma camada especializada de *meta-scheduling* implementada sobre o Kubernetes. Ela serve como a interface primária de desenvolvimento interativo do *cluster*. Através do Run:ai, os pesquisadores contam com

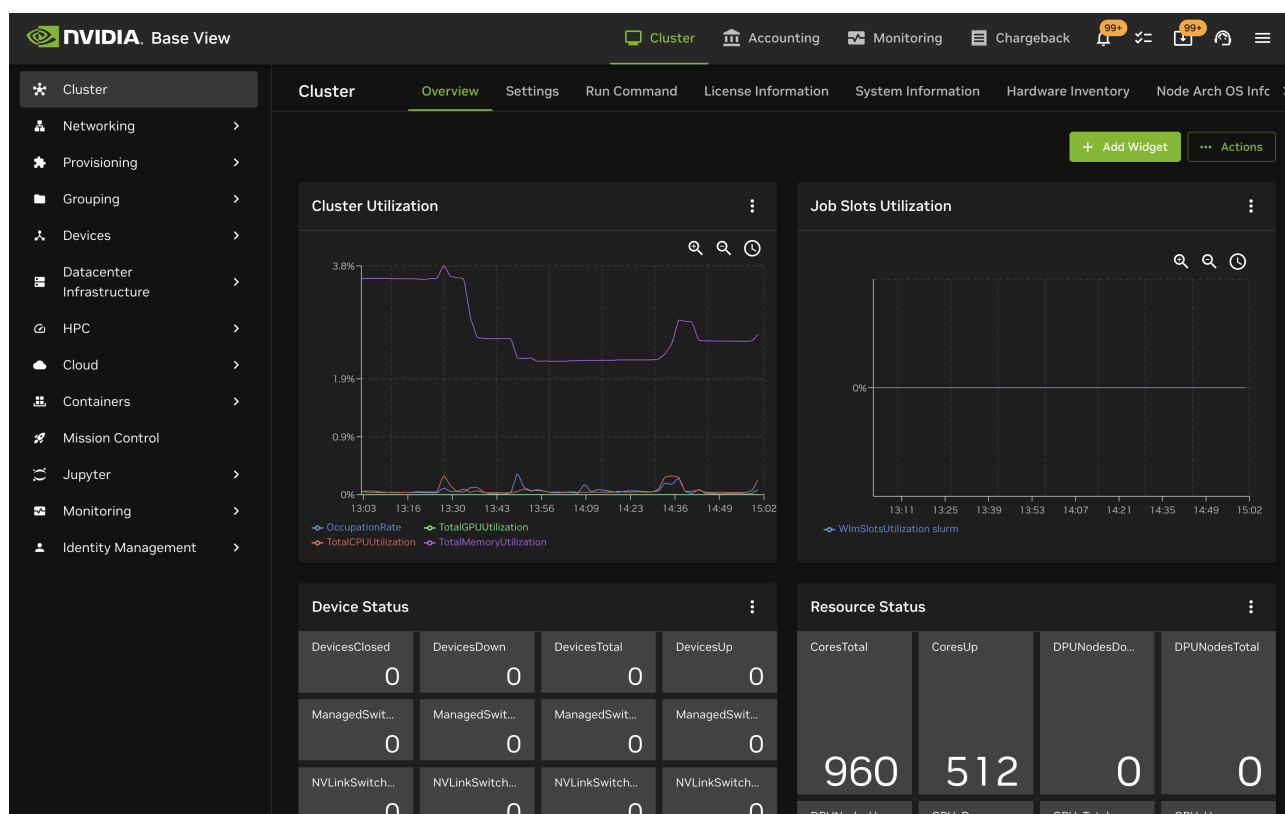


Figura 35: Portal de acesso ao BCM, instalado no *Head Node*.

ferramentas de autoatendimento para alocar dinamicamente frações de GPU e provisionar ambientes como instâncias do *Jupyter Notebook* ou *pipelines* complexos de treinamento distribuído utilizando *frameworks* como *PyTorch* e *TensorFlow*. A plataforma otimiza a alocação de recursos por meio de filas prioritárias, garantindo o máximo aproveitamento do *hardware* compartilhado, como pode ser observado no painel de monitoramento apresentado na Figura 37.

5.3 Próximos Passos

Em decorrência de atrasos acumulados em diferentes etapas após a formalização do processo de compra, principalmente devido à indisponibilidade temporária de cotas de importação no momento da aquisição dos equipamentos, o cronograma original de implantação sofreu impactos significativos. Diante deste novo cenário, e visando mitigar os prejuízos às atividades científicas, pactuou-se conjuntamente com o Instituto Nacional de Telecomunicações (Inatel) a prorrogação do prazo de vigência da fase atual do projeto por um período adicional de seis meses. Essa extensão mostra-se indispensável para readequar o cronograma de pesquisa, garantindo tempo hábil para que as equipes de pesquisadores possam realizar o uso efetivo da plataforma computacional objeto deste relatório. Dessa forma, assegura-se a execução integral dos testes, simulações e entregáveis previstos na fase atual do projeto, maximizando o retorno científico sobre a infraestrutura adquirida.

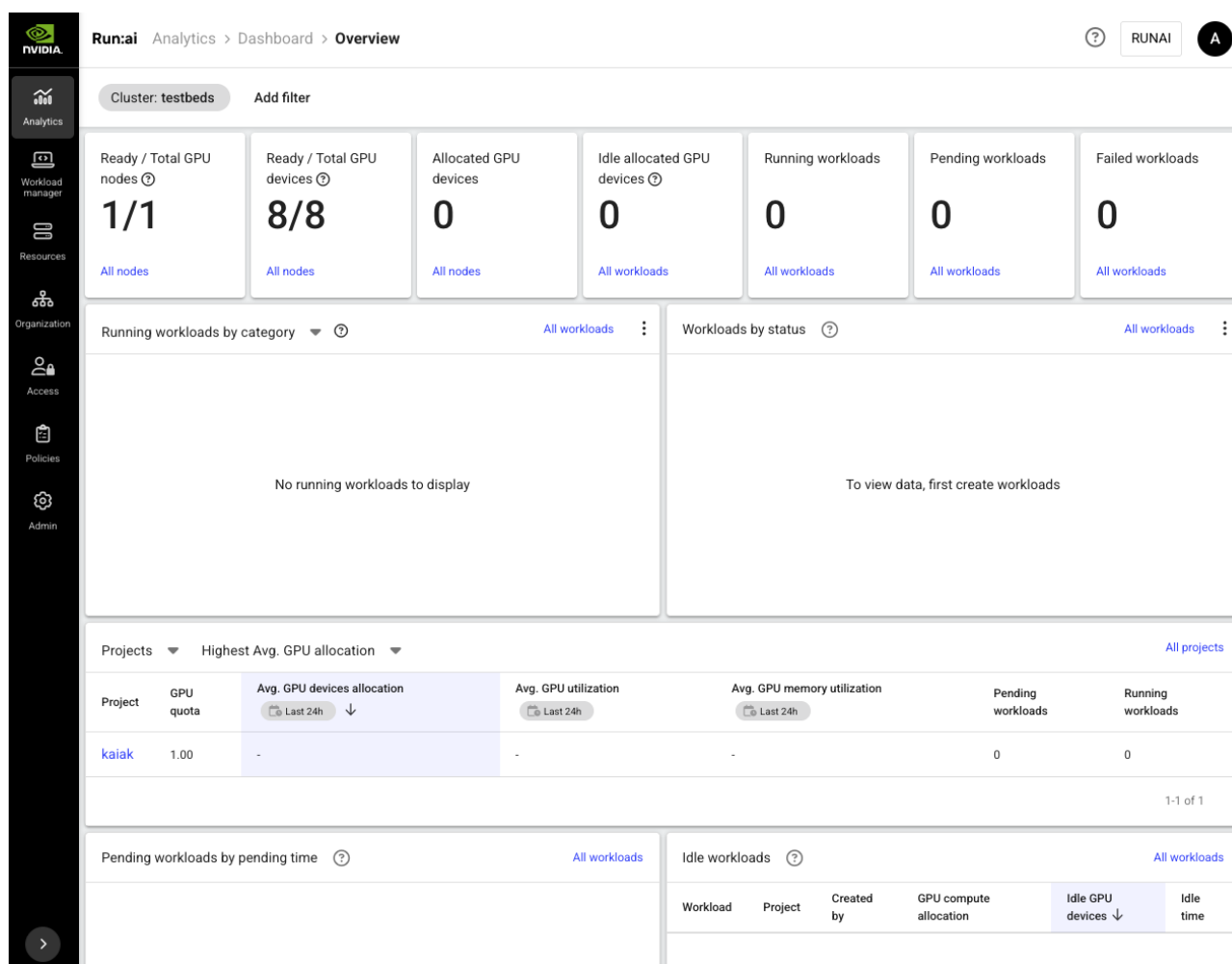


Figura 37: Portal de acesso ao Run:ai, com 1 GPU *node* disponível.

5.3.1 Lições Aprendidas

A execução desta etapa do projeto consolidou aprendizados práticos e estratégicos fundamentais para otimizar futuras aquisições e implantações de infraestruturas computacionais. As principais lições estão detalhadas a seguir.

No âmbito da compra, é importante que o processo de aquisição de equipamentos de alta tecnologia seja iniciado, preferencialmente, no ano anterior ao da execução planejada. O objetivo é garantir que a abertura formal da solicitação de compra ocorra estritamente entre janeiro e fevereiro. Essa janela temporal é crítica para evitar o esgotamento precoce das cotas de importação, garantindo que o cronograma do projeto não sofra impactos devido a uma paralisação do processo de compra.

Com relação à infraestrutura necessária para a instalação de tais equipamentos, é necessário pontuar que demandas de refrigeração e fornecimento elétrico de sistemas de alta densidade são fatores críticos de sucesso. Todos os parâmetros técnicos (como dissipação térmica em *British Thermal Units* (BTUs)/h, redundância de alimentação, capacidade de carga por rack, padrão do cabeamento) devem ser rigorosamente levantados, validados e preparados de forma antecipada, sob risco de inviabilizar o comissionamento físico dos servidores.

Por fim, pensando no aspecto de aplicação e serviço, é preciso pontuar que plataformas

baseadas puramente em soluções *open source* possuem elevado valor estratégico, eliminando o aprisionamento tecnológico (*vendor lock-in*) e oferecendo grande flexibilidade. No entanto, o planejamento deve computar de forma realista o alto esforço interno exigido para a sustentação, integração e atualização contínua desse ambiente pela equipe técnica.

Em contrapartida, a adoção de uma stack integrada de *software* e *hardware* fornecida pela NVIDIA oferece alto grau de automação e simplificação no gerenciamento do *cluster*. Essa abordagem reduz drasticamente a complexidade operacional nas etapas iniciais, viabilizando um cenário de disponibilização e entrega de valor aos pesquisadores de maneira muito mais ágil.

5.3.2 Próximas POCs: Monitoramento, Bilhetagem e Autenticação Federada

Com a base fundacional validada e os componentes de processamento em operação, a trajetória de evolução do ambiente requer a consolidação das camadas de observabilidade e governança financeira, além da integração de mecanismos de autenticação federada. Para as próximas iterações do ecossistema, definem-se as seguintes frentes de trabalho:

1. Monitoramento e Observabilidade: o objetivo é integrar os exportadores nativos da infraestrutura NVIDIA (DCGM), que já coletam métricas granulares de temperatura, consumo energético e eficiência dos *Tensor Cores*, a uma malha de ferramentas *open-source* consolidadas, como Prometheus e Grafana. Isso permitirá a criação de *dashboards* em tempo real cruzando métricas de infraestrutura com eventos de orquestração do Kubernetes e *logs* de processamento do *Slurm*.
2. Bilhetagem e *Accountability*: os próximos esforços focarão no rastreamento preciso do tempo de CPU/GPU consumido por *jobs* alocados em filas de processamento do *Slurm*, e paralelamente no Run:ai serão avaliadas ferramentas de alocação de custos e relatórios de consumo por *namespace*, garantindo transparência e uso racional das GPUs compartilhadas entre os diversos usuários.
3. Autenticação Federada e Gestão de Identidades: visando centralizar o controle de acesso e simplificar a experiência do usuário, a próxima etapa consistirá na integração do ecossistema computacional a provedores de identidade via protocolos seguros como o *Security Assertion Markup Language* (SAML) 2.0 ou o *OpenID Connect* (OIDC). Essa implementação viabilizará o *Single Sign-On* tanto para o acesso interativo à plataforma Run:ai quanto para a submissão de tarefas via *Command-Line Interface* (CLI) no *Slurm*, unificando a gestão de credenciais e garantindo que as políticas de permissão, cotas e controle de acesso baseado em funções *Role-Based Access Control* (RBAC) sejam aplicadas de forma consistente em todo o *cluster*.

6 Conclusão

A implantação da infraestrutura de HPC representa um dos pilares estruturantes do projeto Brasil 6G, uma vez que disponibilizará à comunidade de pesquisadores um ambiente computacional capaz de suportar experimentos de elevada complexidade, treinamento de modelos de IA, processamento intensivo de dados e avaliação de desempenho de tecnologias voltadas às futuras redes de comunicação. Nesse contexto, a infraestrutura transcende o papel de um conjunto de recursos computacionais, constituindo uma plataforma compartilhada de pesquisa que favorece a colaboração entre instituições, a otimização do uso dos recursos públicos investidos e a aceleração do desenvolvimento científico e tecnológico no âmbito do projeto.

Nesta etapa, os esforços concentraram-se na camada lógica da solução, responsável por transformar a capacidade física da infraestrutura em serviços efetivamente utilizáveis pelos pesquisadores. O estudo realizado permitiu contextualizar o panorama das principais tecnologias empregadas em ambientes modernos de HPC, computação em nuvem, além de detalhar a arquitetura de referência adotada para o projeto. Complementarmente, as provas de conceito executadas em ambiente de prototipação possibilitaram validar, em condições controladas, opções tecnológicas de componentes de *software* de mecanismos de orquestração de recursos, assim como avaliar os fluxos de implantação de aplicações e os modelos de compartilhamento do ambiente entre múltiplos usuários.

Os resultados obtidos nesta etapa reforçam a adequação da arquitetura de referência previamente definida para atender aos requisitos levantados junto aos grupos de pesquisa participantes do Brasil 6G. A validação prática da pilha tecnológica demonstra que a solução selecionada reúne os elementos necessários para oferecer um ambiente integrado de computação científica, execução de cargas de trabalho de IA e gerenciamento compartilhado dos recursos computacionais, preservando características essenciais como escalabilidade, segurança, flexibilidade e facilidade de administração.

A fase subsequente do projeto será dedicada à integração entre os componentes de *hardware* e *software*, à configuração dos serviços de suporte e à realização dos testes integrados de aceitação e validação em ambiente de produção. Em razão dos atrasos decorrentes do processo de aquisição dos equipamentos, especialmente aqueles relacionados à indisponibilidade temporária de cotas de importação, a fase atual do projeto possivelmente será prorrogada. Essa readequação do cronograma permitirá que a infraestrutura seja plenamente implantada e efetivamente utilizada pelas equipes de pesquisa, assegurando a realização dos experimentos, simulações e demais atividades científicas previstas. Ao final desse processo, o HPC será disponibilizado à comunidade de pesquisadores do projeto Brasil 6G, consolidando uma infraestrutura estratégica para o desenvolvimento de pesquisas em computação de alto desempenho, inteligência artificial e redes móveis de próxima geração, com potencial para ampliar a capacidade nacional de inovação e geração de conhecimento nessas áreas.

Referências

- [1] B. C. d. Nascimento, M. H. B. Julidori, L. G. B. Guedes, J. d. S. Mata, J. S. Ferreira, and L. L. Mendes, “Projeto brasil 6g fase iii – atividade 4.5: Implantação e disponibilização de infraestrutura de hpc - parte 1,” Rede Nacional de Ensino e Pesquisa (RNP) and Instituto Nacional de Telecomunicações (Inatel), Tech. Rep., jul 2025. [Online]. Available: <https://inatel.br/brasil6g/documents/brasil6g-meta-4-atividade-4-5.pdf>
- [2] NVIDIA Corporation, “NVIDIA AI Enterprise: Plataforma de Software Nativa na Nuvem,” <https://www.nvidia.com/pt-br/data-center/products/ai-enterprise/>, 2026, acesso em: 9 jun. 2026.
- [3] V. O. Dryha, “Using slurm workload manager for managing supercomputers and linux clusters,” *Telecommunications and Information Technology*, 2023.
- [4] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, “Theory and practice in parallel job scheduling,” in *Job Scheduling Strategies for Parallel Processing*. Springer, 1997, pp. 1–34.
- [5] A. B. Yoo, M. A. Jette, and M. Grondona, “Slurm: Simple linux utility for resource management,” in *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003 Seattle, WA, USA, June 24, 2003 Revised Papers 9*. Springer, 2003, pp. 44–60.
- [6] R. L. Henderson, “Job scheduling under the portable batch system,” in *Job Scheduling Strategies for Parallel Processing: IPPS’95 Workshop Santa Barbara, CA, USA, April 25, 1995 Proceedings 1*. Springer, 1995, pp. 279–294.
- [7] S. Zhang, T. Zhang, X. Xie, and W. Zhu, “Feature-driven dynamic hybrid scheduling in slurm,” in *2025 6th International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*. IEEE, 2025, pp. 845–852.
- [8] B. Wu, M. Hu, S. Qin, and J. Jiang, “Research on fusion scheduling based on slurm and kubernetes,” in *International Conference on Algorithms, High Performance Computing, and Artificial Intelligence (AHPCAI 2024)*, vol. 13403. SPIE, 2024, pp. 476–485.
- [9] T. Sterling, M. Anderson, and M. Brodowicz, *High Performance Computing: Modern Systems and Practices*. Morgan Kaufmann, 2017.
- [10] M. A. Jette and T. Wickberg, “Architecture of the Slurm workload manager,” in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Klusáček, J. Corbalán, and G. P. Rodrigo, Eds., vol. 14283. Cham: Springer, 2023, pp. 3–23.
- [11] M. Gkogkas *et al.*, “Running cloud-native workloads on hpc with high-performance kubernetes,” *arXiv preprint arXiv:2409.16919*, 2024.
- [12] A. Nassar *et al.*, “A comparative study of high-performance computing job schedulers,” *Journal of Grid Computing*, vol. 20, no. 4, pp. 1–25, 2022.
- [13] B. Magro *et al.*, “Hpc and cloud convergence beyond technical boundaries,” *Computing in Science & Engineering*, 2024.

- [14] G. M. Kurtzer, V. Sochat, and M. W. Mwenda, “Singularity: Scientific containers for mobility of compute,” *PloS one*, vol. 12, no. 5, p. e0177459, 2017.
- [15] D. Godlove *et al.*, *Apptainer User Guide*, The Linux Foundation, 2023, acesso em: 9 jun. 2026. [Online]. Available: <https://apptainer.org/docs/user/main/>
- [16] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, “Cloud-native computing: A brief review,” in *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2017, pp. 1–7.
- [17] A. Azab, I. Kholod, M. Nasreldin *et al.*, “Enabling docker containers for high-performance and many-task computing,” in *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2017, pp. 279–285.
- [18] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, omega, and kubernetes,” *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.
- [19] H. Liu, X. Wang, and C. Wang, “A cloud-native heterogeneous resource scheduling platform supporting gpu scheduling,” in *Proceedings of the 2025 8th International Conference on Software Engineering and Information Management*, 2025, pp. 159–164.
- [20] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, “Analysis of large-scale multi-tenant gpu clusters for dnn training workloads,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 2019, pp. 947–960.
- [21] Volcano Authors, “Volcano: A Cloud Native Batch System for Kubernetes,” Documentação Oficial, 2026, acesso em: 9 jun. 2026. [Online]. Available: <https://volcano.sh/docs/>
- [22] SchedMD, “Slinky: Slurm on Kubernetes,” Repositório GitHub, 2026, acesso em: 9 jun. 2026. [Online]. Available: <https://github.com/SchedMD/slinky>
- [23] Run:ai Labs, *Run:ai Technical Architecture and GPU Orchestration*, Run:ai, 2024, acesso em: 9 jun. 2026. [Online]. Available: <https://docs.run.ai/>
- [24] G. Symeonidis, E. Nerantzis, A. Kazakis, and G. A. Papakostas, “Mlops-definitions, tools and challenges,” in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2022, pp. 0453–0460.
- [25] D. Kreuzberger, N. Köhl, and S. Hirschl, “Machine learning operations (mlops): Overview, definition, and architecture,” *IEEE Access*, vol. 11, pp. 31 866–31 879, 2023.
- [26] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, “Jupyter notebooks-a publishing format for reproducible computational workflows.” in *ELPUB*, 2016, pp. 87–90.
- [27] E. Bisong, “Kubeflow and kubeflow pipelines,” in *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*. Springer, 2019, pp. 571–585.
- [28] Alnoda Community, “Alnoda Workspaces: Flexible container-based development environments,” Documentação Oficial, 2026, acesso em: 22 jun. 2026. [Online]. Available: <https://docs.alnoda.org/>

- [29] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data,” in *Artificial intelligence and statistics*. PMLR, 2017, pp. 1273–1282.
- [30] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, “Federated learning: Challenges, methods, and future directions,” *IEEE signal processing magazine*, vol. 37, no. 3, pp. 50–60, 2020.
- [31] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, J. Fernandez-Marques, Y. Gao, L. Sani, K. H. Li, T. Almeida *et al.*, “Flower: A friendly federated learning research framework,” *arXiv preprint arXiv:2007.14390*, 2020.
- [32] NVIDIA Corporation, *NVIDIA FLARE: Federated Learning Application Runtime Environment*, NVIDIA, 2024, acesso em: 9 jun. 2026. [Online]. Available: <https://nvflare.readthedocs.io/>
- [33] —, *NVIDIA Base Command Manager 11: Manual*, NVIDIA, 2025.
- [34] FutureLab – Departamento de Ciência da Computação, UFMG, “Flautim: Plataforma de Aprendizado Federado,” <https://flautim.futurelab.dcc.ufmg.br/>, 2024, desenvolvida no âmbito da Linha VI – Conectividade Veicular do Programa Mover. Acesso em: 9 jun. 2026.
- [35] NVIDIA Corporation, “Nvidia nvlink and nvswitch: The building blocks of the advanced data center,” NVIDIA, Whitepaper, 2022.
- [36] —, *NVIDIA Data Center Best Practices: DGX SuperPOD*, NVIDIA, 2024, acesso em: 11 jun. 2026. [Online]. Available: <https://docs.nvidia.com/nvidia-dgx-superpod-data-center-best-practices-with-dgx-b200.pdf>
- [37] R. Buyya, *High Performance Cluster Computing: Architectures and Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999, vol. 1.
- [38] C. Dunigan and The MUNGE Developers, “MUNGE (MUNGE Uid 'N' Gid Emporium),” Repositório GitHub, 2026, acesso em: 13 jun. 2026. [Online]. Available: <https://github.com/dun/munge>
- [39] Cloud Native Computing Foundation, “CNCF Cloud Native Definition v1.0,” Página Oficial da CNCF, 2026, acesso em: 15 jun. 2026. [Online]. Available: <https://github.com/cncf/toc/blob/main/definition.md>
- [40] The Kubernetes Authors, *Multi-tenancy in Kubernetes: Concepts and Best Practices*, Cloud Native Computing Foundation, 2026, acesso em: 15 jun. 2026. [Online]. Available: <https://kubernetes.io/docs/concepts/security/multi-tenancy/>
- [41] Volcano Authors, “Volcano: A Cloud Native Batch System for High-Performance Workloads,” Repositório GitHub, 2026, acesso em: 15 jun. 2026. [Online]. Available: <https://github.com/volcano-sh/volcano>
- [42] Proxmox Server Solutions GmbH, *Proxmox Virtual Environment: Administration Guide*, 2026, acesso em: 15 jun. 2026. [Online]. Available: <https://pve.proxmox.com/pve-docs/pve-admin-guide.html>

- [43] H. Bayraktar, “How to install Kubernetes cluster on Ubuntu 22.04 step-by-step guide,” Medium, 2023, acesso em: 9 jun. 2026. [Online]. Available: <https://hbayraktar.medium.com/how-to-install-kubernetes-cluster-on-ubuntu-22-04-step-by-step-guide-7dbf7e8f5f99>
- [44] Flannel Authors, “Flannel: A network fabric for containers,” Repositório GitHub, 2026, acesso em: 9 jun. 2026. [Online]. Available: <https://github.com/flannel-io/flannel>
- [45] NVIDIA Corporation, “NVIDIA Kubernetes Device Plugin,” Repositório GitHub, 2026, acesso em: 9 jun. 2026. [Online]. Available: <https://github.com/NVIDIA/k8s-device-plugin>
- [46] —, *NVIDIA Container Toolkit Documentation*, NVIDIA, 2026, acesso em: 9 jun. 2026. [Online]. Available: <https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html>
- [47] Volcano Authors, “Volcano Device Plugin,” Repositório GitHub, 2026, acesso em: 9 jun. 2026. [Online]. Available: <https://github.com/volcano-sh/devices>
- [48] The Helm Authors, *Helm Documentation: The Package Manager for Kubernetes*, Cloud Native Computing Foundation, 2026, acesso em: 9 jun. 2026. [Online]. Available: <https://helm.sh/docs/>
- [49] Project Jupyter Contributors, *Zero to JupyterHub with Kubernetes*, 2026, acesso em: 9 jun. 2026. [Online]. Available: <https://z2jh.jupyter.org/>
- [50] The Kubeflow Authors, *Kubeflow: The Machine Learning Toolkit for Kubernetes*, 2026, acesso em: 9 jun. 2026. [Online]. Available: <https://www.kubeflow.org/docs/>
- [51] NVIDIA Corporation, *NVIDIA Base Command Manager 11: Installation Manual*, NVIDIA, 2025, acesso em: 9 jun. 2026. [Online]. Available: <https://docs.nvidia.com/base-command-manager/11.0/installation-manual/index.html>
- [52] —, *NVIDIA Multi-Instance GPU (MIG) Architecture and User Guide*, NVIDIA, 2026, acesso em: 22 jun. 2026. [Online]. Available: <https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html>
- [53] —, *NVIDIA Base Command Manager 11: Kubernetes Integration Guide*, NVIDIA, 2025, acesso em: 22 jun. 2026. [Online]. Available: <https://docs.nvidia.com/base-command-manager/11.0/kubernetes-manual/index.html>
- [54] —, “Nvidia dgx h200 datasheet,” NVIDIA, Datasheet, 2024, acesso em: 20 jul. 2026. [Online]. Available: <https://resources.nvidia.com/en-us-dgx-systems/nvidia-dgx-h200-datasheet>
- [55] Flannel Authors, “Flannel: A network fabric for containers, designed for Kubernetes,” <https://github.com/flannel-io/flannel>, 2026, versão v0.28.6. Acesso em: 20 jul. 2026.
- [56] Presidência da República, “Lei nº 8.010, de 29 de março de 1990: Dispõe sobre importações de bens destinados à pesquisa científica e tecnológica, e dá outras providências,” https://www.planalto.gov.br/ccivil_03/leis/1989_1994/l8010.htm, 1990, acesso em: 27 jul. 2026.

- [57] Sociedade Brasileira para o Progresso da Ciência and Academia Brasileira de Ciências, “Em nota técnica, ABC e SBPC apontam limitações da cota de importação e propõem medidas estruturais,” <https://portal.sbpcnet.org.br/noticias/em-nota-tecnica-abc-e-sbpc-apontam-limitacoes-da-cota-de-importacao-e-propoe-medidas-estruturais> May 2026, publicado em 4 maio 2026. Acesso em: 27 jul. 2026.
- [58] F. Garcia and S. Smaili, “Orçamento de CT&I na LOA 2026: recomposição parcial, equilíbrio necessário e o futuro do sistema científico brasileiro,” <https://portal.sbpcnet.org.br/noticias/orcamento-de-cti-na-loa-2026-recomposicao-parcial-equilibrio-necessario-e-o-futuro-do-sistema-cientifico> Sociedade Brasileira para o Progresso da Ciência, Feb. 2026, publicado em 27 fev. 2026. Acesso em: 27 jul. 2026.